Middleware for Efficient Programming of Autonomous Mobile Robots

TECHNISCHE UNIVERSITÄT DARMSTADT

Vom Fachbereich Informatik der
Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte

Dissertation

von

Dipl.-Inform. Dirk Thomas
(geboren in Groß-Gerau)

Referent:	Prof. Dr. Oskar von Stryk
Koreferentin:	Prof. Dr. Monica Reggiani (Universität Padua, Italien)
Tag der Einreichung:	08.10.2010
Tag der mündlichen Prüfung:	19.11.2010

D17
Darmstadt 2011

Contents

1 Introduction 1
1.1 Contribution . . . 2
1.2 Contents . . . 2

2 Background and State of Research 5
2.1 Background from Robotics . . . 5
2.1.1 Mobility . . . 6
2.1.2 Level of Autonomy . . . 7
2.1.3 Platforms and Scenarios Used in This Thesis . . . 7
2.1.4 Robot Control Software . . . 9
2.2 Background from Software Engineering . . . 13
2.2.1 Software Design . . . 13
2.2.2 Unified Modeling Language . . . 14
2.2.3 Design Patterns . . . 14
2.2.4 Concept of Library vs. Framework . . . 15
2.2.5 Human Computer Interaction . . . 16
2.2.6 System Integration . . . 16
2.3 Specific Requirements of Autonomous Mobile Robots . . . 17
2.3.1 Integration of Elements . . . 17
2.3.2 Runtime Efficiency . . . 17
2.3.3 Efficient Multilevel Testing . . . 18
2.3.4 Flexibility and Adaptability . . . 19
2.3.5 Sophisticated Monitoring and Debugging Methods . . . 20
2.3.6 Expansion to Teams of Robots . . . 21
2.3.7 Existing Approaches for Offline Analysis . . . 21
2.4 State of Research Middleware . . . 23
2.4.1 Middleware in General . . . 23
2.4.2 Message-Oriented Middleware . . . 24
2.4.3 Robotics Middleware . . . 25
2.4.4 Existing Middleware for Autonomous Mobile Robots . . . 26
2.4.5 Comparison and Evaluation . . . 29
2.5 Discussion . . . 29

3 Proposed Methodology for Efficient Middleware 31
3.1 Use Cases and Their Requirements . . . 31
3.1.1 Restricted Onboard Resources . . . 32
3.1.2 Debugging and Monitoring . . . 32
3.1.3 Restricted Bandwidth . . . 33
3.1.4 Offline Analysis . . . 34
3.1.5 Teams of Robots . . . 35

3.2 Evaluation . . . 37
3.2.1 Benchmarking Runtime Efficiency . . . 37
3.2.2 Quantifying Usability and Efficiency of Debugging and Monitoring Tools . 40
3.2.3 Keystroke-Level Model . . . 40
3.2.4 Defining the Scenarios . . . 40
3.3 Concepts for Robotic Middleware Investigated in This Thesis . . . 42
3.3.1 Efficient Local Message Exchange for Common Application Layout . . . 42
3.3.2 Integrated GUI . . . 45
3.3.3 Moving Filtering Data to Publisher-Side . . . 46
3.3.4 Offline Analysis . . . 48
3.3.5 Extending to Teams of Robots . . . 50
3.3.6 Influence on Middleware Interface Design . . . 52

4 Efficient Communication Mechanisms 57
4.1 Message Exchange by Reference . . . 57
4.1.1 Interface Design of RoboFrame . . . 57
4.1.2 Consecutive Execution of Components . . . 57
4.1.3 Enabling Different Strategies of Interaction . . . 59
4.1.4 Application to ROS . . . 59
4.2 Throttle Messages at Publisher . . . 61
4.2.1 Exemplary Throttling Options . . . 61
4.2.2 Throttling at Publisher . . . 62
4.2.3 Differentiating Multiple Subscribers . . . 65
4.3 Recording Intrinsic Data on Robot . . . 66
4.3.1 Intrinsic Data . . . 66
4.3.2 Synchronization . . . 67
4.4 Communication Between Multiple Robots . . . 69
4.5 Bridging Messages Between RoboFrame and ROS . . . 71

5 Efficient GUI Tools 75
5.1 Integrated GUI . . . 75
5.1.1 Different Interface Concepts . . . 75
5.1.2 Widget Toolkits . . . 76
5.1.3 Developed Infrastructure . . . 77
5.1.4 Features for Improved Usability . . . 79
5.2 Developed GUI Components . . . 81
5.2.1 Measuring Execution Time and Frequency . . . 81
5.2.2 Generic Runtime Parametrization . . . 82
5.2.3 Image Viewer and Image Processing Analysis . . . 82
5.3 Extension to Multiple Robots . . . 82
5.3.1 Visualizing the World Models . . . 83
5.3.2 Reusing Single-Agent Debugging Tools . . . 84
5.4 Support for Testing, Debugging and Analysis . . . 85
5.4.1 Recording Intrinsic Data . . . 85
5.4.2 Attaching External Data . . . 86
5.4.3 Event-Based Navigation of Data . . . 87

5.4.4 Automated Analysis with Custom Event Generation . . . 88
5.4.5 Augmenting External Video with Intrinsic Information for Manual Review . 89
5.4.6 Automated Comparison of Competitive Algorithms . . . 89

6 Applications and Results 91
6.1 Applications . . . 91
6.1.1 Humanoid Robots . . . 91
6.1.2 Rescue Robots . . . 91
6.1.3 Employment in Teaching . . . 92
6.1.4 Application in Other Institutes . . . 93
6.2 Local Message Performance . . . 93
6.2.1 Messages in Considered Scenarios . . . 93
6.2.2 Measurement of Latency . . . 94
6.2.3 Impact of a Real-Time Kernel under System Load . . . 95
6.2.4 Impact of Reference Passing . . . 97
6.3 Recording of Messages Locally on the Robot . . . 99
6.4 Team Communication . . . 100
6.4.1 Communication on Various Levels . . . 100
6.4.2 Online Observation During Competition . . . 101
6.5 Integrated User Interface . . . 102
6.5.1 Debugging and Monitoring Tools . . . 102
6.5.2 Extension to Multiple Robots . . . 103
6.6 Sophisticated Analysis Tools . . . 104
6.6.1 Debugging Teams of Robots . . . 104
6.6.2 Automated Detection of Known Issues . . . 105
6.6.3 Augmenting External Video . . . 106
6.6.4 Automated Comparison of Competitive Algorithms . . . 106

7 Conclusion 109

8 Zusammenfassung (Conclusion in German) 113

Bibliography 117

Own Publications 125

List of Figures

2.1 Variety of mobile robots: PR2, Quadrotor, Matilda, Khepara, Aibo 6
2.2 Mobile robots of TU Darmstadt participating in RoboCup 8
2.3 Structure diagram of the hardware components of the humanoid robot 8
2.4 Structure diagram of the hardware components of the wheeled off-road vehicle . . 8
2.5 The Sense-Plan-Act cycle of a robot control software 9
2.6 The most important elements of the autonomous humanoid robot soccer application with the exchanged information 11
2.7 Different methods of system integration 17
2.8 Substitution of real hardware with simulation enabling software-in-the-loop testing 19
2.9 User interface of the Interaction Debugger 22

3.1 Overhead of message exchange due to marshaling, memory copy and demarshaling 33
3.2 The subscription to a message bus implies transferring all messages 34
3.3 Recorded messages are fed to the control software bypassing several components . 35
3.4 Overhead when messages are in the first place skipped by the subscriber 35
3.5 Measuring latency of message exchange using unidirectional communication 38
3.6 Measuring latency of message exchange using round trip communication 39
3.7 Avoiding overhead of message exchange due to passing references 43
3.8 The Composite pattern 43
3.9 The Decorator pattern 44
3.10 Missing flexibility points for altering the interaction between the components and the middleware 44
3.11 Reducing number of exchanged messages utilizing a message filter 46
3.12 Throttling amount of messages directly at the publisher-side 47
3.13 Provided meta information enable the publisher to skip unneeded computations . . 47
3.14 Visualization of intrinsic data from multiple robots 51
3.15 Flexibility points between the components and the middleware enable altering the interaction 53
3.16 The Gateway pattern 53
3.17 The Strategy pattern 54
3.18 Dependency injection is used to decouple a component from the specific gateway . 55

4.1 RoboFrame components provide descriptive information of the exchanged messages 57
4.2 Sequence diagram of the message exchange between RoboFrame and a component 58
4.3 Multiple components share the same message storage through proxies 59
4.4 Implications of different throttling configurations 62
4.5 For ROS the throttle configuration is passed through an anti-parallel topic 63
4.6 The classes involved in the procedure of publishing messages in ROS 64
4.7 The custom classes injected into ROS in order to perform throttling 64
4.8 Messages cannot be routed correctly by the message broker when throttling is applied 65
4.9 The message broker uses the extended header for routing the message to only a subset of the subscribers 66

4.10 The dialog to remotely configure and trigger logging functionality directly on the robot . . . 67
4.11 Sequence diagram to determine the clock offset between two hosts . . . 68
4.12 Clock offset calculation and propagation for multiple robots . . . 69
4.13 The calculation of the global world model . . . 70
4.14 A messaging bridge connecting two messaging systems . . . 71

5.1 The GUI infrastructure with dynamically loadable managers for each middleware . 77
5.2 The messages of independent instances of RoboFrame are transparently exchanged through an explicitly established TCP connection between the message brokers . . 78
5.3 The view displays the execution time and frequency of each component . . . 81
5.4 A generic view for the parametrization of arbitrary components at runtime . . . 82
5.5 The image viewer displays the raw images, visualizes the detected objects and enables graphical debugging of the algorithms' internals . . . 83
5.6 The model viewer visualizes multiple different parts of the world model . . . 83
5.7 The central view for specifying the correlation between views and particular robots 84
5.8 Multiple instances of a single view, each communicating with a particular robot . . 85
5.9 The view records arbitrary messages in the GUI and plays back these logfiles afterwards . . . 86
5.10 List of detected events easing navigating through the numerous amount of recorded data . . . 87
5.11 Conditions of the state transitions with insufficient hysteresis lead to oscillating states in the behavior control . . . 89
5.12 The external video is augmented with intrinsic information of the multiple robots . 90

6.1 Different humanoid robots participating at RoboCup . . . 91
6.2 The newly developed four-legged robot . . . 92
6.3 The number of messages exchanged per second inside the robot control software . 93
6.4 The measured latency for local message exchange on a generic kernel . . . 95
6.5 The measured latency for local message exchange using a real-time preempt kernel 95
6.6 Comparison of the latencies for local message exchange with and without a real-time kernel while the system is idle or under load . . . 96
6.7 The latency for exchanging messages with different sizes locally using various middleware . . . 97
6.8 The latency for different message sizes is reduced to a minimum when utilizing pass-by-reference in RoboFrame . . . 98
6.9 The latency for passing references in RoboFrame with and without a real-time kernel while the system is under load . . . 98
6.10 The dynamic role assignment is based on explicit team communication . . . 101
6.11 The view visualizes data from a team of robots simultaneously . . . 103
6.12 Difficulty to identify the source of problems in complex applications . . . 104
6.13 Intrinsic data enabling the identification of the source of a problem . . . 105
6.14 The detection of landmarks in the soccer scenario using two different algorithms under varying lighting conditions . . . 107
6.15 The detected objects of two different implementations are compared and visualized 107
6.16 The ball distance from two different algorithms is compared and visualized . . . 108

List of Listings

4.1 The main function of a ROS node 60
4.2 A ROS node wrapped in a class featuring dependency injection for the node handle 60
4.3 Modified main function injecting a custom node handle into the ROS node 60
4.4 Transformation of a RoboFrame streamable into a ROS message 72
4.5 Channel adapter to accept RoboFrame messages and publish them using ROS . . . 73
4.6 Channel adapter to subscribe for ROS messages and relay them to RoboFrame . . 74

List of Tables

2.1 The differences between a library and a framework 15

3.1 Subset of operations and assigned durations in the Keystroke-Level Model 41

6.1 Successful works performed on top of RoboFrame 92
6.2 The differentiation of small and large messages exchanged in the scenarios 94
6.3 Reduced overhead for local message exchange 99
6.4 Performance impact of recording large messages in the soccer scenario 100

1 Introduction

Autonomous mobile robots nowadays are deployed in many different scenarios. Among the various applications the competitions of RoboCup and of the Defense Advanced Research Projects Agency (DARPA) are some of the most visible events. The international research and education initiative RoboCup has chosen to use soccer as one out of several primary domains. In multiple different leagues various technologies to perform autonomous soccer games are developed and evaluated at annual tournaments. The challenges held by the DARPA have the objective to develop autonomous vehicles capable of driving in traffic and performing complex maneuvers such as merging, passing, parking and negotiating intersections.

Developing and programming of such autonomous robots is a complex, time consuming and error-prone task. A large variety of hardware and software components has to be developed to achieve a well performing robot. Additionally, these parts have to be integrated into an overall system and tested for correctness and robustness thoroughly. At the same time different subsystems are composed in order to test and evaluate only subsets of the complex application. The task of *system integration* becomes increasingly essential in the future, when more and more high level functionality and various different algorithms have to be integrated into more complex autonomous robots.

A *middleware* eases the interfacing of these various components and provides fundamental properties well known from software engineering, e.g., decoupling of software blocks, distributed processing, flexibility, and therefore makes the process of software development more efficient. These criteria apply to any complex software, as it needs to be able to evolve based on changing requirements and objectives.

However, besides these common software engineering aspects, *unique requirements* arise due to the specific hardware used in robotics, especially for mobile robots, and due to the real-time constraints for physical robot actions. Depending on the hardware platform, the payload and therefore the computational resources for real-time performance are quite limited. Consequently, the *runtime efficiency* is of special importance in this domain.

The high degree of autonomy demands for the integration of various sensors. Complex algorithms from different domains are utilized for processing the available data and making decisions based on this information. Each single algorithm as well as the overall system requires *sophisticated debugging and monitoring tools* in order to identify defects in single components as well as problems of the integrated overall system.

Furthermore, the robots' mobility induces a restricted connectivity and bandwidth to external computational systems. This makes remote monitoring and debugging tasks more challenging and demands for additional *offline analysis* capabilities. Especially in the scenario of multiple cooperating autonomous robots in a dynamic environment, the amount of intrinsic information generated during robot operation is tremendous. Analyzing and reviewing such data manually is extremely time consuming and error-prone, and quickly becomes unmanageable and inefficient. Algorithms for *automated analysis* of the information are to be applied in order to make the review process more efficient.

Middleware follows the horizontal system integration paradigm and address most of these demands. Many different robotic specific middleware approaches have been developed in recent years. Depending on the purpose and the field of application these approaches greatly differ

regarding the provided functionality. Most of the current robot middleware provide common communication functionality between the different elements of the control software. While some have a focus on distribution and scalability, others aim for hard real-time support or concentrate on the abstraction of specific kinds of hardware or algorithms. The supported platforms vary from embedded computers with specific operating systems to distributed computation networks with heterogeneous operating systems.

Current robot middleware provide features for specific sets of use cases and scenarios. Often, multiple solutions can be used together as they cover different aspects.

However, almost all of the *existing approaches fall short* when high demands for local runtime efficiency are posed, which is essential for mobile platforms with low computational resources. Also, the support for complex debugging and monitoring tasks is limited or the capabilities are restricted to selected parts of the control software. Especially in the context of *teams* of autonomous mobile robots the existing approaches are limited in analyzing the distributed information jointly.

1.1 Contribution

This work addresses the specific needs of autonomous mobile robots. The efficiency of programming such systems is increased for different phases of the software development process.

On the one hand, the local runtime efficiency is investigated, which is of crucial importance for mobile robots. While providing valuable software engineering properties like decoupling, a middleware always introduces an additional level of abstraction. Naturally any intermediate layer brings along extra efforts and expenses. This *overhead is reduced* to a minimum in cases where efficiency is highly required, while preserving the advantages of the middleware.

On the other hand, the specific requirements for efficient debugging and testing are considered. The focus is on an integrated user interface, which allows efficient and sophisticated online and offline analysis of complex applications. Additionally, several steps of the *analysis process* are *automated* in order to reduce the time spent for these repetitive tasks. Several features are dependent on fundamental functionality of the utilized middleware.

Of particular interest is the application in the domain of *teams of autonomous mobile robots*. Thus, the transfer of procedures and tools from single robots to a team of collaborating robots and their specific demands are investigated.

The presented concepts imply a particular design of the middleware and of the interface with the components. The implementation is based on *RoboFrame*, a custom middleware and software architecture for autonomous mobile robots, co-developed by the author.

Additionally, the concept has been adopted to *ROS*, an emerging open-source meta-operating system for robots. However, the concepts can be transferred to other middleware for the purpose of enabling the same enhancements.

1.2 Contents

An overview of the current state of research is given in Chapter 2. It includes common aspects of robots and their control software and the specific properties of autonomous mobile robots as well as general software engineering disciplines and tools. In particular, the concept of middleware is introduced with a focus on message-oriented middleware for system integration, and various different *existing robotic middleware* are described.

In Chapter 3 a set of requirements for efficient programming of autonomous mobile robots is proposed based on various different use cases. The runtime efficiency is considered for several common applications as well as the flexibility of the application design and adaptability to changing requirements. Furthermore, the required capabilities for efficient testing and debugging of complex applications and teams of robots are outlined. Also, methodologies for evaluating programming efficiency are discussed. Thereupon, the concepts for improving the efficiency of different aspects are described. On the one hand, the overhead for the intra-host information exchange is addressed as well as the efficient reduction of the amount of data required for remote monitoring and offline analysis tasks. On the other hand, the usability enhancements of integrated graphical user interfaces, the efficient applicability to teams of robots and the improvements using automated analysis are conceived.

Based on the proposed concepts an *improved middleware interface* featuring superior runtime efficiency for common use cases is developed. The realization of these concepts is described in Chapter 4. The implementation as part of RoboFrame is explained in detail. Additionally, some concepts are carried over to a second middleware, namely ROS, to demonstrate the general applicability of the design. Finally, the aspect of communication in a team of robots is examined.

Chapter 5 considers the design and development of an *integrated graphical user interface*. The specific challenges of complex applications for autonomous mobile robots are outlined and the implementation of the concepts is illustrated. While the advanced features for *analyzing teams of robots* are all based on RoboFrame, the principle of the integration concept is also carried over to a solution, which is independent of a concrete middleware.

Within this work, the communication infrastructure provided by the middleware has been enhanced and several tools for testing, monitoring and debugging have been developed, implemented and applied successfully for the development of software for teams of autonomous biped, quadruped and wheeled robots. An overview of these tools and their application and evaluation in various scenarios for autonomous mobile robots is given in Chapter 6. The description is complemented with results for a comprehensive set of benchmark problems to testify the presented improvements of the local message exchange systematically and evaluate the developed methodologies.

The thesis closes with a summary and a conclusion in Chapter 7.

2 Background and State of Research

This chapter starts with a brief introduction to robotics and the definition of what kind of machines are called robots. Thereupon the specific characteristics of mobile and autonomous robots are considered. Furthermore, the platforms and scenarios used in the applications of this thesis are presented and the involved elements of their control software are mentioned.

Section 2.2 introduces general software engineering aspects and common instruments, as they form the basic principles of the following considerations. The specific requirements in the context of autonomous mobile robots are outlined in Section 2.3.

Section 2.4 mentions different concepts of system integration and investigates message-oriented middleware further. Thereupon, multiple existing approaches from the robotics domain and their applicability for the specific scenarios are discussed. The chapter closes with a short summary of essential properties of middleware and discusses the differences, assets and shortcomings of the described software.

2.1 Background from Robotics

As strange as it might seem to be, there is really no standard, equally accepted definition for a robot. To quote Joseph Engelberger, a pioneer in industrial robotics: *"I can't define a robot, but I know one when I see one."*

However, the following characteristics are considered characteristically for a robot:

- *Sensing* — a robot needs the ability to sense its surroundings as well as its internals. Therefore, different kinds of sensors can be used, which can be classified, among others, in visual, tactile, position and distance sensors. The most commonly used are cameras (visible as well as infrared), laser and sonic range finders, bumpers, force sensors, position and shaft encoders.

- *Processing* — a robot must have the capability to process data and make choices depending on this information. This feature differentiates a robot from a simple mechanical device and is enabled by a computational unit. This may be any kind of computer or even a micro controller.

- *Physical interaction* — a robot must be able to physically interact with its environment to achieve a specific task. This may cover pick-and-place operations of any kind of object as well as carrying out series of operations like using a tool for welding tasks. Therefore, it needs actuators to move the parts of the mechanical structure. By far the most popular kinds of drives are electric motors and pneumatic and hydraulic actuators. Besides these, various other technologies have been developed but are not widely used: series elastic actuators, shape memory alloy, electro-active polymers, piezoelectric motors and many more.

Therefore, besides a mechanical structure forming the robot's motion system, robots consist of sensors for perceiving the environment, a computational unit, and joints, which can be moved by actuators.

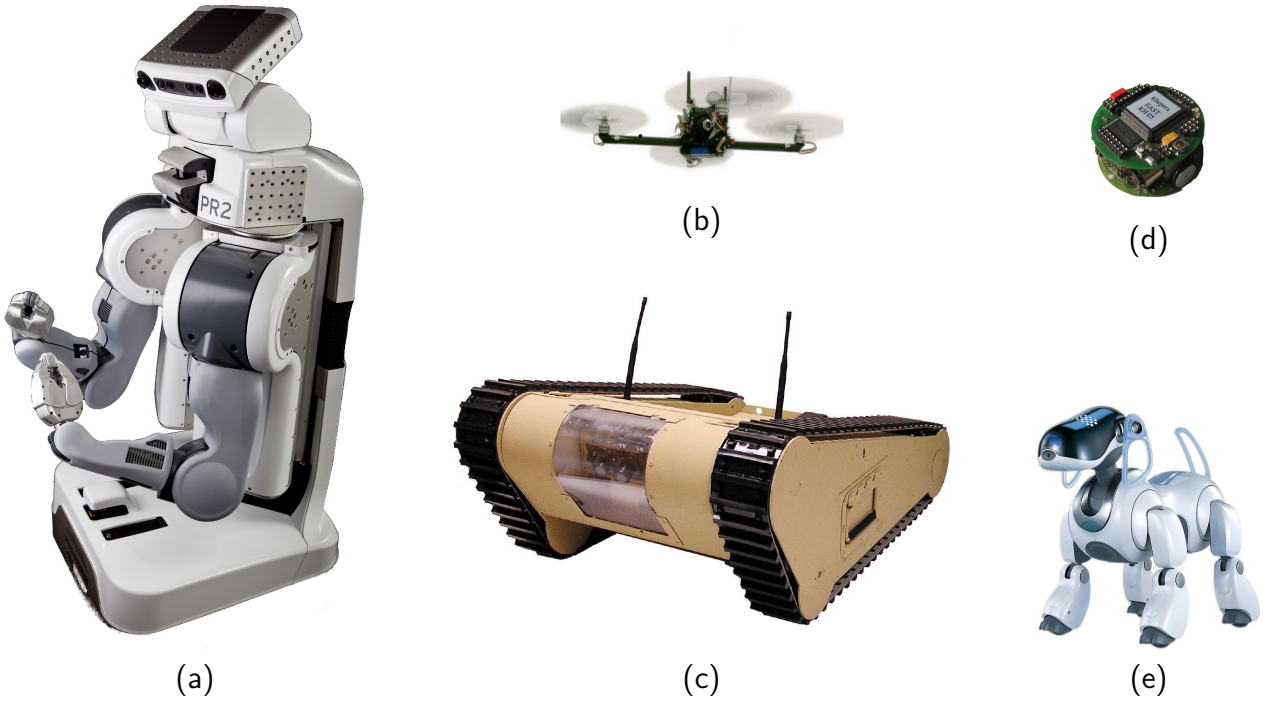


Figure 2.1: Variety of mobile robots: (a) PR2 from Willow Garage, (b) Quadrotor, (c) Matilda from Mesa Robotics, (d) Khepara mobile robot, (e) Sony Aibo

2.1.1 Mobility

In contrast to stationary systems mobile robots are additionally able to move around in the environment, which extends their field of operation. Several tasks are rendered possible only by the capability of locomotion. Various kinds of locomotion can be used depending on the application and especially the characteristic of the ground surface. The most common principles are the usage of wheels and tracks for rolling movements. Another difficult but promising approach uses legs to walk like animals or humans. Other modes are flying and swimming methods or even unusual techniques as snake-like locomotion. An in-depth introduction to robots with a focus on mobility as well as autonomy can be found in [95].

But the advantage of mobility comes with a substantial disadvantage. The *payload* of a mobile platform is *limited*. Hence, the available computational resources are restricted likewise. Additionally, a mobile robot must bring a self-sustaining power supply along, which serves all on-board components especially the actuators. The impact of these limitations on the considered scenarios is described in Section 2.3.2.

The selection of different mobile robots in Figure 2.1 illustrates the *heterogeneity of mobile robots*. Depending on the size and weight of the robot as well as the kind of locomotion the limitations are quite restrictive. The payload of the robots shown in Figure 2.1 (a) and (d) varies significantly. These particular constraints have a direct impact on the kind of computational unit which can be carried. This ranges from a single micro controller to several high-capacity platforms featuring numerous multi-core processors. Likewise differs the network connectivity of (b) and the complex legged locomotion of (e) from the other mobile platforms.

2.1.2 Level of Autonomy

The classification of the level of autonomy takes the interaction between human control and the robot into account. On the one hand, the simplest form is called *teleoperation*. The operator controls every detail including the definition of the mission goals, the navigation of the vehicle as well as the executed motions and avoidance of obstacles. On the other hand, the robot acts *fully autonomous* without involving any human interaction.

Between these two limits any degree of interaction and autonomy is reasonable. A set of levels defines the transfer of more and more authority to the robot [29]. The graduation between these levels is fluent, for example:

- In *safe teleoperation* the robot is responsible for avoiding obstacles autonomously.
- In *standard shared mode* it also takes over the execution of the motions, while the overall navigation is still controlled by an operator.
- In *collaborative tasking mode* the task of navigation is also delegated to the robot, whereupon the operator only provides the objectives.

As several tasks are currently better performed by a human operator one possible solution is to dynamically merge the good qualities of both humans and robots into a single system. This approach is called *sliding or adjustable autonomy*.

The scenarios presented in the following are mostly targeting autonomous operations and therefore involve complex algorithms. These properties increase the efforts necessary to analyze the robots' behavior even further as described later in Section 2.3.5.

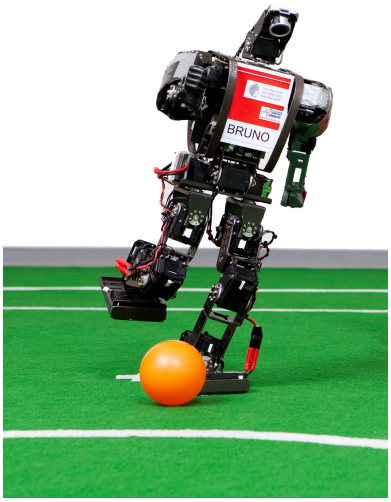
2.1.3 Platforms and Scenarios Used in This Thesis

The author's group is working with a set of highly heterogeneous robots ranging from various wheeled and legged platforms to marine and aerial vehicles with different levels of autonomy.

Two of these robot models are participating at the RoboCup, which is an international research and education initiative. The yearly competitions serve as an environment for providing standardized benchmarks for autonomous robots to foster artificial intelligence and robotics, where a wide range of methodologies and technologies can be examined, integrated and compared. The ultimate goal of RoboCup is to develop a team of *fully autonomous humanoid robots* that can win against the human world champion team in soccer. Mainly the following two scenarios have been used as applications for this thesis.

The *Darmstadt Dribblers* [22] are participating in the Humanoid Kid-Size League since 2004. In this class fully autonomous robots with a human-like skeleton and human-like senses play soccer against each other. The biped robots are actuated by 21 servo motors and have a total height of approximately 60 cm (Figure 2.2a). They feature only human like passive sensors like a directed camera, acceleration sensors and gyroscopes to comply with the rules of the league. The computational power is provided by a micro-controller and an embedded PC using an AMD Geode or Intel Atom processor (Figure 2.3). At the competitions each team consists of three robots playing soccer on a field with color-coded objects. This scenario constitutes a highly dynamic environment, which makes high demands on the reactivity of the robots' behaviors.

The *Team Hector* [98] is participating in the Rescue Robot League since 2009. The task of disaster response involves moving through an unknown area while creating a map and detecting



(a) Humanoid robot of the Darmstadt Dribblers in 2010 (Source: Katrin Binner)

(b) Wheeled off-road vehicle of the Team Hector

Figure 2.2: Mobile robots of TU Darmstadt participating in RoboCup

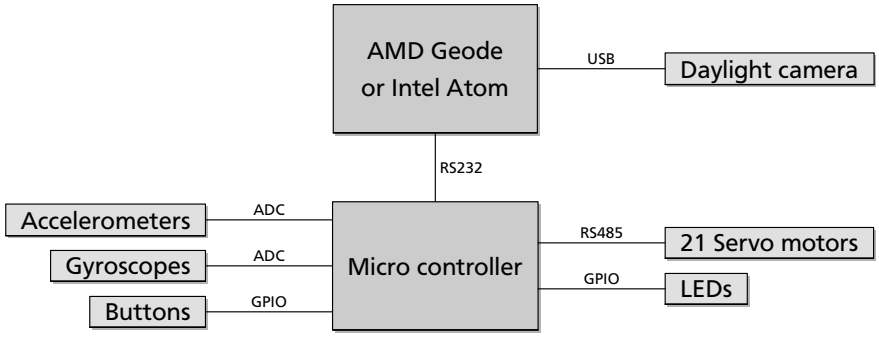


Figure 2.3: Structure diagram of the hardware components of the humanoid robot

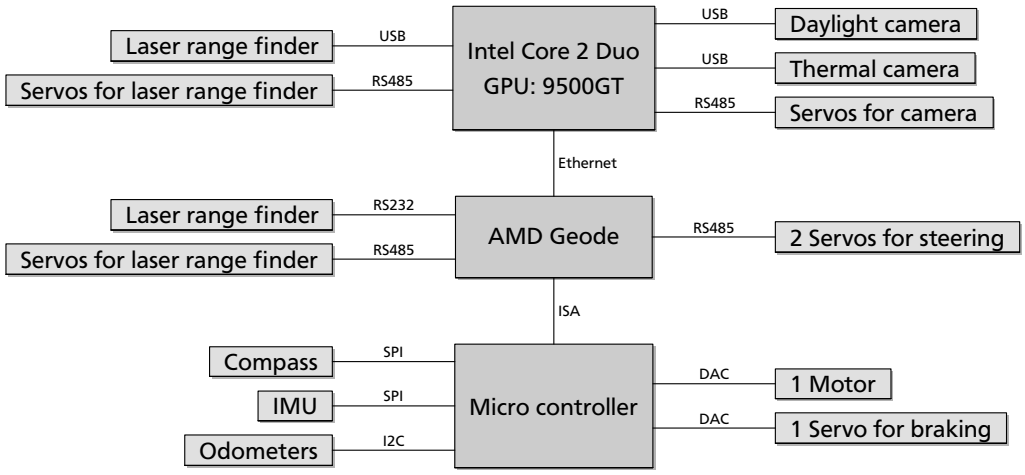


Figure 2.4: Structure diagram of the hardware components of the wheeled off-road vehicle

victims and hazardous objects. The sensors of the four wheeled off-road vehicle (Figure 2.2b) are not restricted in this league, which permits the usage of additional sensors like laser range finders and infrared cameras. Furthermore, two separate computers are used, which are responsible for different calculations. The detailed structure of the hardware components is depicted in Figure 2.4. The level of autonomy varies between the different missions from teleoperated to fully autonomous.

Especially, the hardware platform used in the first scenario - the humanoid robot - is considered lightweight. Therefore, the constraints to computational power are much more severe than in the second scenario as well as compared to other robots, which are also considered mobile but are based on a fundamental different level of hardware (e.g. the PR2 of Willow Garage).

2.1.4 Robot Control Software

The software, which runs on the computational unit and operates a robot, is called robot control software. It is responsible for processing the data provided by the sensors, deciding on the next actions to achieve a specific goal and controlling the actuators. The concept of control can be seen schematically as a continual process of the three steps: sense, plan and act (Figure 2.5) following the characteristics of robot mentioned in Section 2.1.

The sense-plan-act paradigm appears on multiple different levels within a control software as described in [3]. For example, on the lower level a control algorithm is used for every motor to actuate the individual joints of the robot depending on the current and the target position. On the higher level the behavior control decides in which direction a mobile robot should move based on information provided from the models of the environment, the robot's state and a given target.

Behavior Paradigms

To determine the behavior of a robot several different approaches have been investigated as described in [72]. *Reactive systems* [11] directly use the input of the sensors to determine the actions the robot should take next. This approach limits the planning of the next action to be based

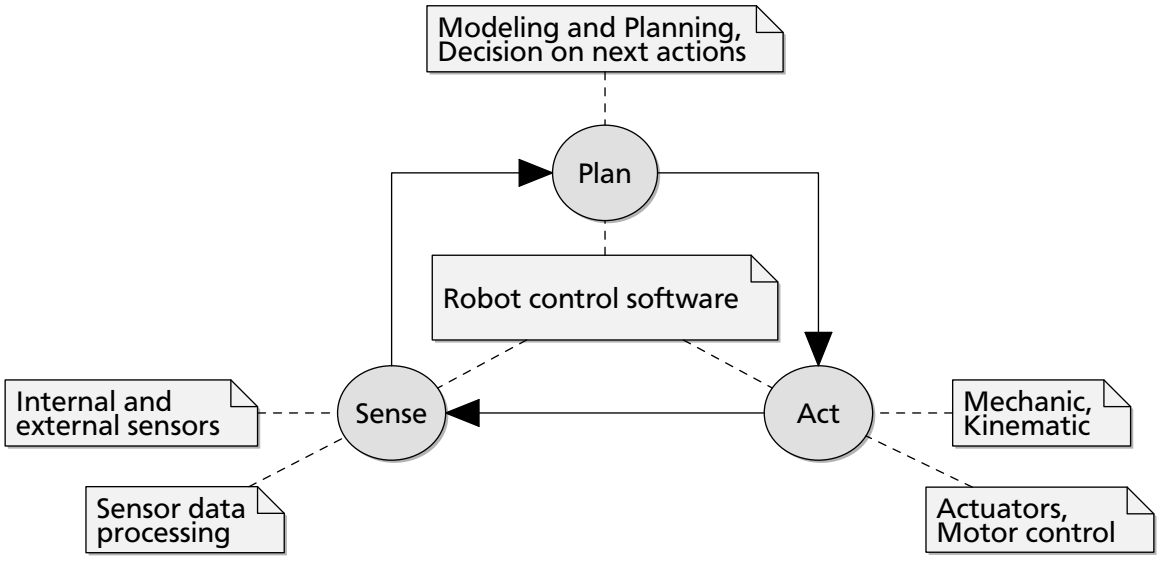


Figure 2.5: The Sense-Plan-Act cycle of a robot control software

on the current sensor information only. It allows for immediate reactions to rapid changes in a dynamic environment. Hence, the usage is limited to applications, which do not require any long term planing.

In contrast *deliberative systems* are based on an internal representation. This representation includes any kind of information about the environment and is called the world model. Thus, in this planning step the world model is updated using the available sensor data. Afterwards, the decision of the robot's actions are made based on that model instead of the raw sensor data. The model can provide functionalities like filtering and include information of former sensor data, it is tailored for long term planing. Indeed, it is not well suited for reacting in a rapidly changing environment.

Hybrid systems [7] combine the advantages for strategic planing (from deliberative systems) and reactive behavior (from reactive systems).

Elements of a Robot Control Software

Every robot control software consists of multiple elements, which all suit a specific need in the sense-plan-act cycle. For any of these tasks different approaches and algorithms are reasonable. Figure 2.6 depicts a subset of the elements and the exchanged information in the robot control software used in the described robot soccer scenario. In the following commonly applied methods and techniques from the various specific domains are described briefly.

Sensor Processing

Each kind of sensor provides a specific type of data, which may be either intrinsic or extrinsic. Thus, the amount of data as well as the frequency varies vastly. E.g. an accelerometer provides only some bytes per reading but it can provide measurements with a high frequency of several thousands per second. In contrast, a camera can deliver megabytes of raw data per frame with just a few tens of frames per second.

Sensor processing is the first step in the sense-plan-act cycle. Its task is the extraction of specific features of interest from the raw sensor readings. Depending on the environment the sensor processing can be used to detect abstract features or known objects, as in the highly structured environment of the soccer scenario.

Due to the various kinds of sensor types an extensive amount of different processing approaches exist. Even in the same domain multiple different algorithms are developed and applied, each with a different focus and specific assets and drawbacks.

In the domain of computer vision, various algorithms are applied to extract any kind of feature, e.g., the color-coded objects in the described soccer scenario as well as human faces or people. In Section 6.6.4 different algorithms with specific advantages are compared. A comprehensive overview on the different methods can be found in [23]. In an unknown environment only abstract features can be detected, e.g., finding corners in scans from a laser range finder or feature like SIFT [70].

All measured data are represented within the particular coordinate frame of the related sensor. But commonly the subsequent modeling steps require the extracted features to be expressed in an egocentric coordinate system of the robot or in absolute world coordinates. Thus, the sensor processing can also provide a transformation of the information to different coordinate systems. This operation requires additional knowledge, namely the transformation matrix between the sensor's mount point and the origin of the target coordinate system. If these two are not rigidly

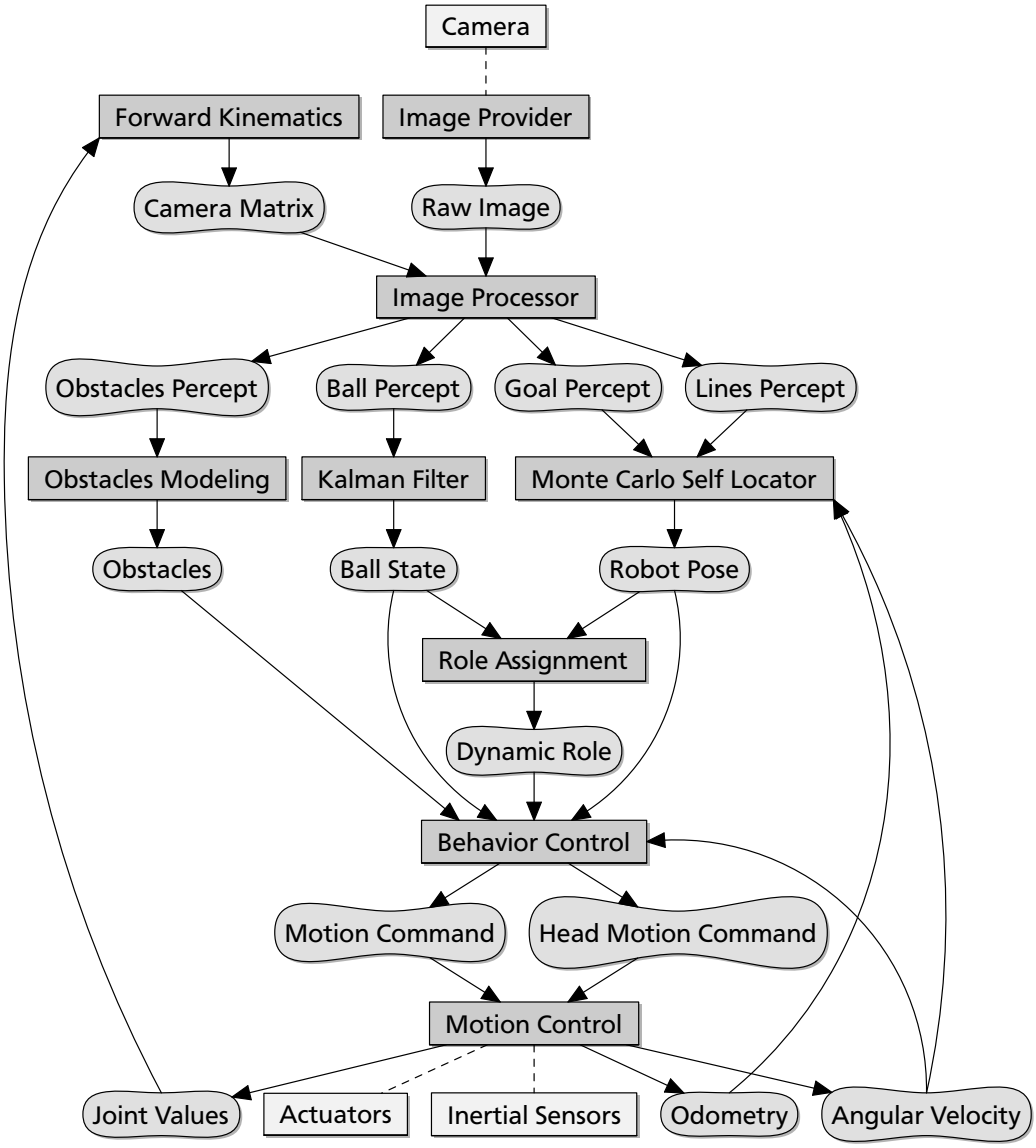


Figure 2.6: The most important elements (rectangular) of the autonomous humanoid robot soccer application [22] with the exchanged information (rounded)

connected but across one or multiple joints, the transformation matrix must be dynamically computed using forward kinematics [21].

World Modeling

The world modeling maintains information of the environment as well as of the robot's state. The model is updated using the data provided from the preceding step of sensor processing. Commonly, information provided from multiple sensors are fused into a combined consistent model. But it may also involve multiple independent representations like a global map containing the robot's pose besides an egocentric description of obstacles.

For the different tasks, like self localization and tracking specific objects in the environment, various different algorithms exist. Common approaches use filtering methods (e.g. Kalman-Filter [56]) and probabilistic algorithms (e.g. Monte-Carlo algorithm [33]) to maintain and update the model information. In an unknown environment, where no map of the surroundings is available,

the map must be built in parallel to exploring and self localization. This task of simultaneous localization and mapping is abbreviated with SLAM [26, 27] and is adopted for the described rescue scenario [62].

When multiple robots should act cooperatively in a team to achieve a common goal, they need to exchange information or even share a joint model. This can be realized in different ways: using a central or decentral approach. On the one hand, a common world model is centrally updated and shared between all teammates. Such an approach demands reliable access to the central data pool. On the other hand, each robot is self-contained and maintains its own local model. It incorporates information received from the teammates [17, 93]. Indeed, in this case it must be ensured that the distributed information is accurate enough to achieve a joint goal.

Behavior Control

The task of the behavior control is to decide which actions the robot should execute next. The decision may be based on either the world model, directly on the sensor data, or a combination of both according to the aforementioned behavior paradigms.

Many different techniques have been developed over the years, starting with the classical subsumption architecture [12]. Other methods use hierarchical finite state machines [68], behavior descriptions based on petri-nets [107] or reinforcement learning strategies [97]. A comprehensive overview on the recent methods can be found in [85].

Path Planning

The goal of the path planning is to find a valid route to a specific destination. This may rely either on an available map or only utilize the sensors to decide between free space and obstacles. A comprehensive overview of developed methods can be found in [67].

In the mentioned rescue scenario, the A* (A star) search algorithm has been adopted for the pathfinding task, which is described in [41].

Motion Control

The motion control is responsible for controlling the actuators in order to perform the desired motion. Depending on the kind of locomotion system of the robot, different methods are applied.

For wheeled robots the velocity of each wheel as well as the steering needs to be determined and controlled. Many different variations for a number of actuated and non-actuated wheels as well as for the steering exist. One approach for a four-wheel differentially driven mobile robot can be found in [15].

For legged systems the determination of the trajectories of each limb becomes more complex due to the numerous degrees of freedom, which need to be coordinated. Common approaches especially for robots with at least four legs use a central pattern generator from which the individual positions of each leg and joint are derived. Other methods are inspired by animals and replicate reflexive control concepts. Details about an approach combining these two concepts can be found in [61].

To maintain the stability of legged robots additional techniques are applied, which utilize criteria for static and dynamic stability, e.g., the zero-moment-point theory [35]. In addition, the generation of the legged motions can be adjusted by an extensive set of adjustable parameters. To find parameters, which entail optimal gaits and satisfy stability and velocity objectives, optimization techniques can be applied. Details about the optimization method applied in the above-mentioned soccer scenario can be found in [42].

2.2 Background from Software Engineering

The topic of software engineering [25] is divided in multiple sub-disciplines ranging from requirement analysis over software design, development, testing and maintenance, to management of the configuration, engineering, development process and quality.

2.2.1 Software Design

The software design describes both the *architectural view* and the *algorithmic implementations*. Many different aspects must be considered when designing a software like a middleware. Some of the most important aspects are:

- *Extensibility* — the future growth of the software is taken into consideration and it should be easily possible to integrate new capabilities without the need to alter existing elements or even the underlying architecture.
- *Modularity* — to separate implementation and testing of different aspects of a software, it should comprise of several well defined but independent components, which leads to better maintainability.
- *Reusability* — to enable the reuse of single components each should be limited to the essence of the functionality expected.
- *Usability* — this criterion for a software highly depends on its target audience and may vary from ease of learning for developers to convenient user interfaces for end users.

Depending on the specific characteristics of an application, the importance of each single aspect must be carefully considered, since some goals are contrary.

In the context of this thesis, the following other design aspects have a lower relevance:

- *Compatibility* — it should permit interoperability with other products.
- *Fault-tolerance* — in case of errors it should withstand or even be able to recover.
- *Robustness* — it should be robust in case of unusual or unexpected input and be able to operate under heavy load.
- *Security* — the software should resist malicious interferences and not disclose secret data to unauthorized entities.

Reusability

The aspect of reusability will be detailed exemplary. A comprehensive work of reference on modern software design can be found in [10].

To achieve the goal of building increasingly complex applications in a reasonable time and quality, reusability is a key feature. Reusability is the likelihood that a software can be used again with at best no modification in another context.

The benefit of software reuse is versatility [90]. The number of fixed errors increases from reuse to reuse improving the quality of the software. Likewise the productivity is increased since less

code has to be developed and tested. Without the reuse of significant portions of the software from the soccer scenario the period for entering the rescue league would have been significantly longer as described in Section 6.1.2. For a detailed definition of the different aspects of the product quality in software engineering see ISO/IEC 9126 [52].

Admittedly, the advantages of software reuse are not achievable without a penalty. Developing software with reuse in mind requires additional efforts, skills and foresight during design, implementation and testing. But the initial learning effort for reusing other software is easily compensated by the increased productivity in the long-term. Furthermore, the performance of the implementation is improved from reuse to reuse just as the quality of the software.

In the context of both described scenarios, the relevance of reusability is significant, as the major part of the control software as well as the graphical tools supporting the development process are shared between both applications.

Opposing Objectives

Obviously, a specifically engineered software tailored for one particular use case is very likely better suited and more efficient than a software developed with generic reuse in mind. In this spot the demand for reusability from the software engineering view impedes the requirement for efficiency stated as a software quality criteria.

Similarly, it is difficult to balance between the opposing objectives of *loosely coupling* for better maintainability, reusability and *tightly coupling* for more efficient and scalable integration as described in [13, p. 31-70]. This problem reappears on multiple different levels, e.g., at the integration level, when considering the usage of a middleware as well as when defining the interfaces for heterogeneous hardware and software.

It is necessary to ponder on the importance of both demands and base the decision on the aimed scenario. The duty of the software design is to foster the reusability of components, while keeping the overhead due to the decoupling low, so that the eventually reduced efficiency does not become a counter-argument.

2.2.2 Unified Modeling Language

To describe the structure and behavior of a software in a generic format, the Unified Modeling Language (UML) has been developed by the Object Management Group (OMG) to express information and knowledge and provide an explicit visual representation. In this thesis, many design concepts and implementation details are illustrated by UML diagrams for clarification and better comprehension. The semantics of the entities and relations in the graphs are explained in [8, 31].

2.2.3 Design Patterns

When designing software, some common problems reoccur over and over again. Several design problems are not even domain specific, but reappear in several different contexts.

A well known description of patterns came from Christopher Alexander. Even though the origin is from a different area, namely the architecture, his description characterizes the meaning of patterns very well:

"Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution of that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [1].

A design pattern describes an elegant solution to a specific but common problem in software design. It captures the essence of the approach in a succinct and easily applicable form. Yet these patterns are not directly usable source code fragments. They have to be reapplied for every specific application. Still, when reapplied in a future development, it benefits from the utilization and review in the past.

Therefore, design patterns provide a way to reuse design solutions that have been proven to work in the past. Additionally, they make the communication between developers easier due to a commonly known vocabulary of pattern names.

The book about design patterns from the Gang of Four [39] describes the most fundamental patterns subdivided into the three areas of creational, structural and behavioral patterns. Related works specialize on patterns for enterprise application architecture [30] or especially to the domain of messaging solutions [48]. Several of these design patterns are used in the following chapters.

2.2.4 Concept of Library vs. Framework

Since many of the later presented solutions are realized either as a library or a framework, the differences between these two approaches must be stated clearly.

A library is a collection of classes or subroutines. In contrast a framework is an abstraction providing generic functionality, which can be overridden or specialized by custom code with specific functionality. It consists of both *reusable code* and *reusable design*. To express the differences to libraries, some striking characteristics are compared in Table 2.1.

The following features distinguish a framework from a library:

- *Inversion of control* — the flow of control of a program is not specified by the caller but by the framework
- *Default behavior* — the framework provides a reasonable default behavior
- *Extensibility* — the user extends the framework by selective overriding or specializing with custom code

A comprehensive overview of frameworks and their relation to designing reusable classes can be found in [55].

Table 2.1: The differences between a library and a framework

	A library is ...		A framework is ...
...	about reusable functionalities	...	about reusable behaviors
...	something you call/inherit from your code	...	something that calls your code or provides services for your code
...	a collection of classes or components	...	how abstract classes and components interact with each other

2.2.5 Human Computer Interaction

The requirements on the interface for human computer interaction (HCI) depend highly on the targeted user. In this thesis and the described scenarios the users are all developers in the domain of robotics working on different aspects of the robot control software. But in general the users may have substantially different levels of previous knowledge and skills, which make designing an interface applicable for all user groups more challenging.

Additionally relevant to the concrete requirements are the actual use cases performed using the interface. In this context this may range from monitoring and remote control tasks to in-depth testing and debugging of specific subsystems of the robot control software. In many cases, graphical visualizations are required to make the voluminous amount of different information clearly represented.

In many cases a single interface very likely does not fit all use cases and user groups well, even if graphical user interfaces are the de-facto standard and essential for visualization tasks. Other kinds of interfaces are equally feasible. For example, software developers are familiar to employing command line interfaces, as they permit a faster way of interaction and the ability to automate sequences of reoccurring actions.

Thus, different interfaces are of varying effectiveness depending on the tasks to be accomplished. A comprehensive overview of the general topic of HCI can be found in [24].

For the domain of user interface design similar patterns and techniques have been developed as for classical software engineering. A comprehensive set of extensively described patterns for interface design can be found in [102].

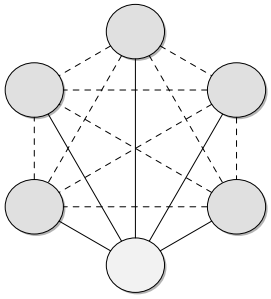
2.2.6 System Integration

For the following the *terminology* of a *component* is clarified. It does not follow the strict definition, which can be found in [19]. Particularly, a component must not necessarily be a binary unit of deployment, but the concept of using a component in an unmodified fashion still holds.

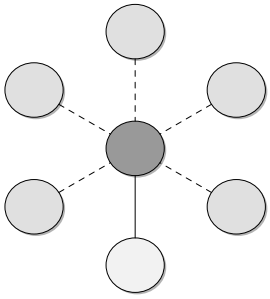
Nevertheless, after a set of components has been designed according to the above-mentioned criteria of extensibility and modularity and with reusability and usability in mind, they have to be linked together. This process of composing different applications and software components into an overall system is called system integration. Different integration methods can be applied and are categorized as follows:

- *Vertical integration* — integrates subsystems directly according to their functionality. This method is specific to a particular use case and is not reusable for different purposes. Admittedly, it is the approach with the lowest effort in the short term.
- *Star integration* (Figure 2.7a) — integrates any subsystem by linking it to every other directly. Therefore, new subsystems need to be integrated with every existing subsystem. This approach, while looking promising at first, does not scale well.
- *Horizontal integration* (Figure 2.7b) — introduces a dedicated subsystem, e.g. a middleware, which is responsible for handling the communication between other subsystems. Thus, any subsystem must only interface the communication subsystem and none of the others.

Due to the benefits in the long run, only the concept of horizontal integration is investigated further.



(a) Star integration: each component is linked with every other



(b) Horizontal integration: a dedicated subsystem is introduced with which each connection is linked

Figure 2.7: Different methods of system integration

2.3 Specific Requirements of Autonomous Mobile Robots

Complementary to the general aspects, in the following, several specific requirements are founded based on the above-mentioned common characteristics of autonomous mobile robots.

2.3.1 Integration of Elements

As outlined in Section 2.1, many elements are involved in a robot control software for an autonomous robot.

On the one hand, the numerous elements should be developed as independent components, each from the specific developers well skilled in their particular domain. Dependencies between these components must be avoided wherever possible, as a clear separation-of-concern leads to an improved testability of subsystems and a better maintainability in the future.

On the other hand, a multitude of elements needs to be integrated into a coherent overall system. Ideally the composition of the elements poses no overhead compared to a monolithic built system.

These two goals are *conflicting*. But due to the extensive advantages of a modular software, an overhead introduced through the separation is widely accepted. Otherwise the development of complex robot control software becomes unmanageable. For complex applications like autonomous robots the horizontal integration method is applied, because of the better scalability in the long term. The thereby introduced indirection in the form of a dedicated communication layer naturally comes with the trade-off of extra efforts at runtime.

In the future it is certain that the computational resources will continue to increase, both in the quantity of CPU cores and in the number of dedicated computers. Hence, the communication layer eases the distribution across multiple CPUs as well as multiple computers to scale with the hardware advancements.

2.3.2 Runtime Efficiency

For most non-robotic applications the overhead of the integration method is negligible. Their overall load is not high enough to make the communication performance an issue or even a bottleneck [44]. But mobile robots present a special case.

As any mobile platform has a more or less restricted payload, the available battery capacities as well as the computational power are limited. Hence, the robot control software must keep its used CPU resources and with it the power consumption as low as possible. Often the robot control software utilizes all available resources. Consequently, any kind of overhead, as little as it may be, needs to be revised. Above all, the overhead must not lead to decisions against other valuable software design criteria.

This is especially valid for lightweight mobile robots as the constraints are even tighter.

2.3.3 Efficient Multilevel Testing

However, the software of autonomous robots comprises multiple complex algorithms. Testing an overall robotic system is the final step in the development. Otherwise it is more feasible to test only subsets of components simultaneously and separate the tests into multiple smaller tasks.

In robotics and especially for mobile robots the information available for the decision process is afflicted with a lot of uncertainties. They comprise any kind of sensor data, which transforms continuous analog information from the real world to discrete digital values. In contrast to many pure software projects, the input data are by no means perfectly accurate or even free of errors. Additionally, many uncertainties occur during the interaction of the robot with its environment. A common issue for mobile robots is for example the inaccuracy of the locomotion due to slipping and friction.

Thus, each application needs to be reduced as much as possible in order to avoid influences from other components as well as any kinds of uncertainties, which are inherent in robotic system. For a lot of test scenarios it is permissible to remove several components or replace them with simple stub implementations. A common technique is to replay previously recorded sensor data in order to test and debug some of the algorithms using a known set of input data.

But this approach is not limited to a specific level in the robot control software. In can be applied for any kind of test as long as no closed loop is required. As soon as the results of one cycle need to be fed back for the next cycle it is necessary to use a simulator.

Testing Using Simulation

The component of the robot control software interfacing the real sensors and actuators can be replaced with complements, which access a simulator to provide sensor data and simulate the interaction through the actuators. This method as depicted in Figure 2.8 enables *software-in-the-loop testing*.

The simulation can be adjusted to either provide perfectly accurate sensor data or emulate noise input data. The same applies to the interaction with the environment. Depending on the test either an ideal locomotion without slipping and friction can be simulated or a more realistic model can be utilized.

While most frequently applied to sensors and actuators, the simulator can also provide information without uncertainties on any other level. For example, the simulator can provide ground-truth data like the position and orientation of all robots as well as all relevant objects in the simulation [80].

Even if simulation cannot fully replace tests with the hardware in a real-world scenario, it is essential to the development process. On the one hand, it eases the testing and debugging of

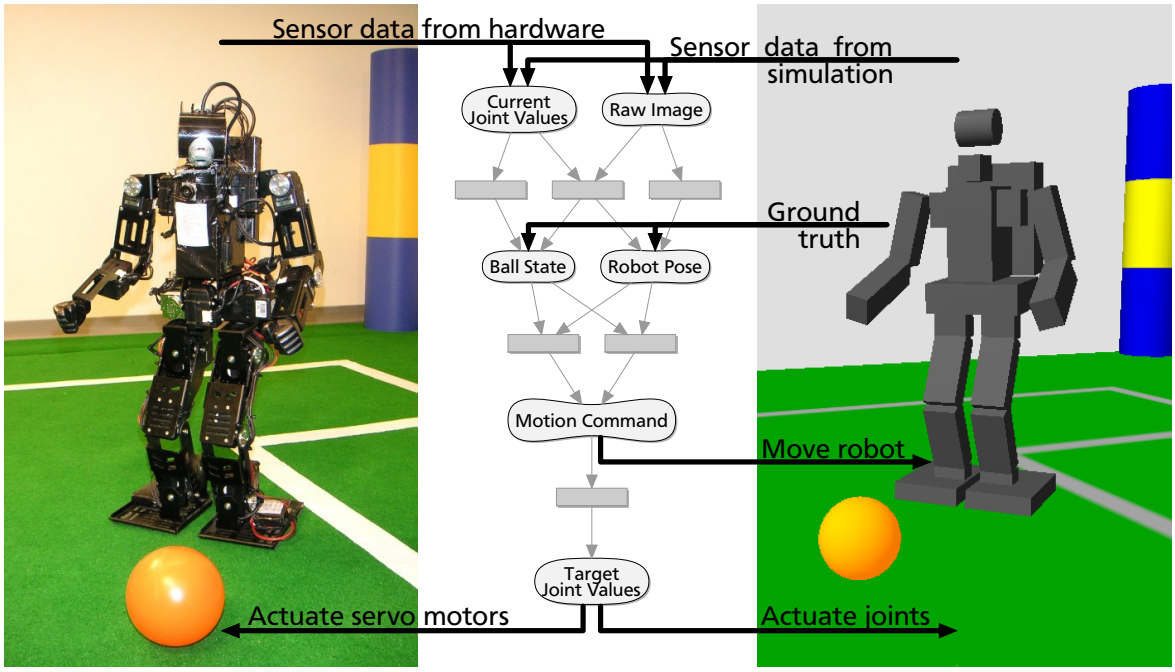


Figure 2.8: Substitution of real hardware with simulation enabling software-in-the-loop testing

specific elements of the control software. On the other hand, which is at least as important, the utilization of simulations instead of real world testing circumvents wear and potential damage to the robot's hardware. Additionally, the ability to run tests independent of the hardware permits an arbitrary number of tests, which could even be executed in parallel.

2.3.4 Flexibility and Adaptability

As described above, a robot control software for autonomous mobile robots performing extensive tasks forms a very complex system. Many different elements are involved in the process of collectively reaching a specific goal. Each single element can have arbitrary complexity and typically deals with domain specific problems.

In order to enable the previously described testing of subsets of components, the application layout must be adaptable. The composition may be different for each single test scenario and must therefore be customizable efficiently. Additionally, the scenarios and tasks are ever changing and new innovations in hardware, techniques and algorithms are rapidly being developed. Thus, it must be possible to easily replace single elements (e.g. the ones depicted in Figure 2.6)with alternative developments to address changes of hardware interfaces and incorporate newly developed methods. Such modifications must be *transparent* to the rest of the software wherever possible to keep the efforts of implementation low.

Besides the inherent complexity of robotics systems and the challenges of distributed environments, the demand for flexibility is one of the most important criteria. A detailed view on component interoperability and flexible wiring between those is given in [13, p. 183-210].

2.3.5 Sophisticated Monitoring and Debugging Methods

The complexity of the control software and the mobility of the robot also affects the monitoring and debugging capabilities. Several specific properties make the tasks of monitoring and debugging more challenging compared to pure software projects.

Due to the mentioned uncertainties repeated test runs are unfortunately not perfectly deterministic. Rather minor variations in the sensor data or the interaction with the environment might change the outcome of a test series significantly. Therefore, the described multilevel testing strategies are crucial in order to debug efficiently.

But several other issues arise due to the specific characteristics of autonomous mobile robots as described below.

Visualization and Interaction

In the process of monitoring such complex robot control software, various information of the control software must be readily available to provide a complete view of the robot's internal state. This involves visualizing raw sensor readings, illustrating the current state of modeling the environment and keeping records of the decision process of the behavior control. Furthermore, most algorithms can be parametrized, which pose an extensive set of possible adjustments and options to intervene.

Depending on the use case, the most efficient way of interaction may vary significantly. While for some tasks a graphical user interface is mandatory and suitable, for others it might not be effective. Especially, developers are used to command line interfaces and value the quick interaction in order to automate common operations. Consequently, the user interface must support different paradigms of user interaction to suit the varying demands.

Offline Analysis

Additionally, several properties of autonomous mobile robots make online analysis complicated or even unfeasible.

On the one hand, the limited communication abilities between the mobile robot and a stationary monitoring instance hinders extensive monitoring jobs. A limitation in the bandwidth between these can render complex monitoring tasks as well as classic debugging approaches impossible. Thus, alternative procedures must be applied to gain in-depth knowledge of the complex system. One viable solution is to defer complex debugging tasks past the test run and conduct the analysis subsequently.

On the other hand, several issues are unfeasible to monitor and trace online because of the frequency of the changes or the amount of relevant information. In order to be able to debug and monitor these cases thoroughly, offline analysis is a mandatory feature. In both cases all relevant information needs to be logged on the robot and be made available for later detailed analysis and debugging.

Even with these capabilities to analyze the information offline, the task of reviewing the data is extremely time consuming and error-prone. Hence, it is desirable to provide algorithms which ease the review process automate several subtasks and therefore make the analysis more efficient. This especially applies, when a team of collaborating robots is analyzed.

2.3.6 Expansion to Teams of Robots

The enlargement of the scenario to multiple collaborating robots results in another dimension of complexity.

All of the previously described challenges increase in complexity with the involvement of teams of robots. The aspect of the collaboration makes the applications of multiple robots dependent on each other. The communication, coordination and distribution between the robots introduce another set of uncertainties. Furthermore, the existing graphical tools for visualizing and monitoring are commonly only suited to work with a single robot. Instead of reinventing such functionality for teams of robots they need to be reused to keep the effort low. For the offline analysis, the distributed recorded information from multiple robots needs to be synchronized for reasonable evaluation.

Test scenarios in the real world involving multiple robots become increasingly difficult to manage. For software-in-the-loop testing the required efforts are much lower. Therefore, the aspect of testing in simulation becomes more important.

Special attention must be payed to the efficiency of robot simulation. The required computational power to simulate multiple robots and concurrently execute multiple instances of the robot control software is tremendous. On the one hand, the simulator may reduce the accuracy of the simulation quality in favor of less utilized resources (Section 2.3.3). On the other hand, it should be possible to distribute the multiple instances of the robot control software over several computers.

Demand for External Reference Data

As a result of the increased number of robots, the amount of information relevant for the analysis easily becomes overwhelming. In more complex situations it is even impossible for a human operator to evaluate the correctness and accuracy of the intrinsic information. In order to enable the user to objectively judge the quality of the recorded information an additional reference is required. This external reference information is mainly provided for the manual review process and must not necessarily be interpreted automatically.

2.3.7 Existing Approaches for Offline Analysis

In the following, existing approaches which address the challenging demands for offline analysis are presented, in order to reveal still outstanding features for efficient analysis of teams of autonomous mobile robots. Two of these are also incorporating external reference data for improved offline analysis.

LogPlayer and Vizard

The team *The Ulm Sparrows* participated in the Middlesize League of RoboCup. Their sophisticated logging and offline analysis tool *Vizard* is described in [104]. Although these tools permit significant improvements in different areas, several fields of application are still to be addressed. For example, the synchronized playback for distributed recorded data and the integration of external reference information are crucial for teams of autonomous robots acting in a highly dynamic

environment. Furthermore, the amount of data, which needs to be recorded for the considered scenarios in order to conduct comprehensive offline analysis, is considerably higher as described in Chapter 6.

GermanTeam LogViewer

A solution from the *GermanTeam* of the former four-legged Standard Platform League of RoboCup is the *LogViewer* application [85]. It was developed in order to analyze and debug the complex team behavior of the robots. The approach integrated an external video to provide additional information to the developer for analyzing the recorded data. The intrinsic data of each robot are logged locally without any centralized infrastructure. Likewise, the external video could be recorded using a common video camera. The drawback of this tool is the limitation of the analysis to the behavior control. Other aspects of the control software cannot be reviewed using this application.

Interaction Debugger

An existing solution from another domain is the *Interaction Debugger* [63]. The software collects data from the robot *Robovie* [51] as well as external information from the environment, particularly audio and video. This information can later be analyzed to enable fine-grained inspection of human-robot interaction (Figure 2.9). The drawback of this approach is, that the synchronized recording requires a central capturing computer, which makes it unsuitable for mobile robots.

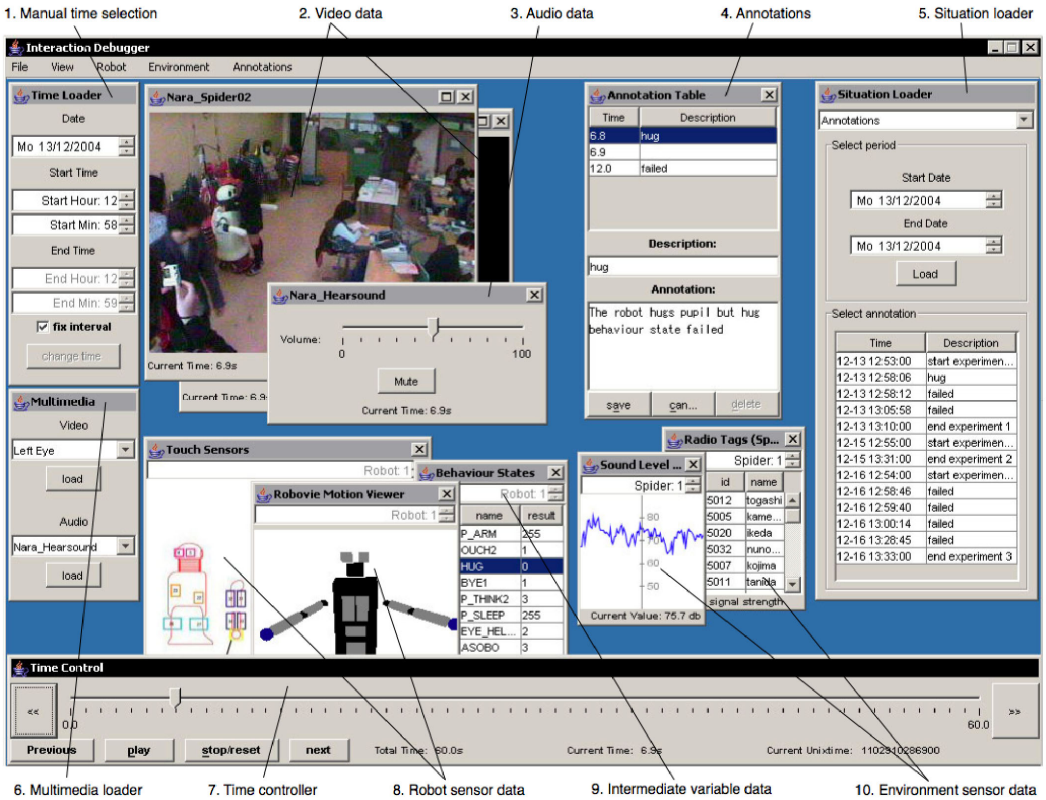


Figure 2.9: User interface of the Interaction Debugger (Image taken from [63])

2.4 State of Research Middleware

2.4.1 Middleware in General

Middleware follows the horizontal integration method presented in Section 2.2.6 and mediates interaction between multiple software components or applications. It introduces an extra component in the software architecture of an application, which provides mechanisms to exchange information between them. As middleware systems rely on interception and indirection mechanisms, they inevitably induce performance penalties.

Since the fields of application are manifold, the demands on and the characteristics of a middleware are divergent too. The following categories are used to differentiate the various kinds of middleware:

- *Transactional middleware* (TM) or transaction processing (TP) monitor [4] — is designed for distributed asynchronous transactions. It is responsible for coordinating transactions between processes and always keeping the overall system in a consistent state. It is mostly applied to handle load balancing operations.

- *Procedural middleware* (PM) or remote procedure calls (RPC) — allows the execution of procedures located on a remote system as if they were local. The communication is typically synchronous.

- *Object-oriented middleware* (OOM) — evolved from RPCs and extends them from procedural to object-oriented concepts. It permits the transparent usage of remote objects. Whereas synchronous communication is common, asynchronous object requests are also possible.

- *Message-oriented middleware* (MOM) — is based on exchanging messages between communication endpoints. While it usually supports both synchronous (via message passing) and asynchronous (via message queuing) communication, the latter is more ofter utilized.

Detailed descriptions, an evaluation of criteria like reliability, scalability and heterogeneity, and specific advantages and disadvantages of these different kinds of middleware can be found in [82].

The most visible solutions are Java's Remote Method Invocation (RMI), Microsofts Component Object Model (COM) and OMGs Common Object Request Broker Architecture (CORBA) [45] specification with its various different implementations, e.g., The ACE ORB (TAO) [83]. All three solutions belong to the group of RPC middleware. While RMI and COM are tied to a single platform, namely Java and Microsoft Windows respectively, CORBA provides real cross-platform support as it is independent from the programming language as well as the operating system. For CORBA, the interfaces of the RPC stubs are specified using the Interface Definition Language (IDL) in a language independent way. This enables the Object Request Brokers (ORB) to be implemented in any programming language. The General Inter-ORB Protocol (GIOP) provides the interface to link together various different ORBs.

In the following, the group of MOM is contemplated as it offers specific features especially useful in the domain of robot control software.

2.4.2 Message-Oriented Middleware

The messages represent the common basis for endpoints to exchange information. Apart from that, the endpoints are fully isolated from each other. This independence leads to the desired software design goal of *full decoupling*. Additionally, the messages can be easily recorded and replayed for testing or subsequently processed or analyzed, which is especially relevant for testing complex autonomous robot control software. The *message broker* [48, p. 322-326] is responsible for forwarding the messages between the endpoints. It can either be realized using a centralized broker or in a decentralized fashion distributed across the smart endpoints. It enables dynamic routing of messages within the middleware layer itself, thus permitting flexible control and adaption of the information flow as demanded in Section 2.3.4.

Like the other types of middleware, MOM features a transparent distribution across multiple systems. Due to the inherent asynchronism, messages are buffered in queues when the destination is busy. Some solutions take this concept one step further and do not imply the sender and receiver to be connected at the same time. Therefore, persistent storage to back up the message transfer medium is required. This becomes especially useful when dealing with unreliable networks and discontinuous connections. On the contrary, some MOM additionally provide a synchronous interface, which encapsulates a request-response pattern and masks the inherent asynchronism from the application.

Messaging Paradigms

Different messaging paradigms can be applied to pass messages between endpoints. Point-to-point, request-response, publish/subscribe and blackboard systems are the most commonly used ones.

For distribution, the blackboard system is less suited since the synchronization of distributed shared memory becomes increasingly complex and resource intensive. Both, the *point-to-point* and the *request-response* communication, are only applicable for two endpoints exchanging information. The *publish/subscribe* paradigm supports multiple publishers as well as multiple subscribers. As subscribers should only receive a subset of the total messages published, the messages need to be filtered.

Two common forms of filtering messages exist:

- *Topic-based* — the publishers classify the messages into topics, which represent logical channels. All involved endpoints connect to a topic specific *message bus* [48, p. 137-141], which forwards the published messages to all subscribed endpoints.

- *Content-based* — in contrast to the previous filtering, the subscribers query the messages based on specific attributes and properties of the data. Therefore, each subscriber formulates constraints and only the messages which match these criteria are delivered to the subscriber.

Independent of the filtering, the publications are commonly implemented using a broadcast or multicast, in order to reduce the utilized bandwidth compared to multiple point-to-point connections. Most MOM support multiple paradigms to provide an adequate solution for the various respective use cases.

Lack of Standards

In the group of OOM, the CORBA standard is implemented by many vendors, which provides clear interfaces through the use of IDL and interoperability based on the GIOP. Until now no comparable solution for MOM exist. The lack of an official standard hinders a wide adoption and obstructs the compatibility of custom solutions. Each custom implementation features its own application programming interface (API), message format and tools for configuration and management.

Several different protocols have been developed. The Extensible Messaging and Presence Protocol (XMPP) [89] employs an XML-based message format. Another example is the RESTful Messaging Service (RestMS) which works over plain HTTP. These solutions are more suitable for web services than for robotics applications.

Java Message Service (JMS) [14] is a standard, which only defines the API of a MOM. But it lacks the definition of a common message format, for this reason different JMS implementations still remain incompatible.

An emerging standard is trying to close this gap. The Advanced Message Queuing Protocol (AMQP) [64] defines various messaging paradigms as well as the formats used for messaging and has reached draft status[1]. Yet, solutions implementing the latest standard have to come up and it is unclear if this standard will be widely adopted in the future.

The disparity is not only related to the communication mechanisms, but recurs on various different levels. Even for the fundamental representation of geometric data no consensus exists across the existing software in robotics as stated in [13, p. 107-124].

2.4.3 Robotics Middleware

The advantages of common middleware are reasonable for complex robotics applications. Their support for decoupling, distribution and heterogeneity is crucial for the scalability and reusability of the developed software in the future.

In the previous section a set of specific requirements has been derived from the considered scenarios of autonomous mobile robots. But the various different applications in the robotics domain have specific demands. For example, the control algorithms of robotic manipulators induce hard real-time constraints on the middleware.

Each subdomain has its own unique requirements, which are not yet addressed by a general-purpose robotics middleware. As an example the scenario of wireless sensor networks (WSN) is described briefly.

Middleware for Wireless Sensor Networks

The goal of a WSN is to monitor an area of interest using a distributed network of sensor nodes. This could involve any kind of physical or environmental conditions, such as motion, pressure, temperature and pollution. The sensor nodes are distributed across the area and usually extremely lightweight, often consisting of a single micro-controller and a radio transceiver only.

Due to the specific hardware constraints and the scenarios, a middleware for WSN has to deal with very restricted energy resources and network connectivity. In these cases features like queuing

[1] AMQP 1.0 Draft as of Mai 19, 2010

and relaying of messages, consideration of connectivity constraints and reduction of communication as much as possible, in order to reduce the utilized energy resources, are significant. The requirements on scalability, distribution and sophisticated routing mechanisms differ significantly from the considered scenarios of autonomous mobile robots.

Therefore the specific requirements of WSN are not considered in this work. More detailed information on middleware for WSN can be found in [40]. The existing robotics middleware considered in the following are therefore limited to the subdomain of autonomous mobile robots.

2.4.4 Existing Middleware for Autonomous Mobile Robots

Due to the innumerable existing approaches for middleware for autonomous mobile robots only a limited set can be mentioned in this thesis.

CLARAty

The Coupled-Layer Architecture for Robotic Autonomy [106, 78, 77] is a framework for integrating, maturing, and validating new technologies developed by institutions and universities under NASA's Mars Technology Program. The architecture is separated into a functional layer, which uses an object-oriented system decomposition, and a decision layer, which is based on a declarative model. One of the main goals is the tight coupling between declarative and procedural-based algorithms. With this design it represents an application framework for a specific class of heterogeneous robots. As a communication layer The ACE ORB (TAO) [92] is utilized, which is a standards-compliant real-time C++ implementation of the CORBA standard based upon the Adaptive Communication Environment (ACE) [91].

MARIE

MARIE [20] is an open-source design tool for mobile and autonomous robot applications developed by the Mobile Robotics and Intelligent Systems Laboratory at the University of Sherbrooke. It integrates existing software using a component-based approach and acts as a central mediator for functional components according to the horizontal integration paradigm. Therefore, numerous different tools are reused, e.g., Player, CARMEN and ACE, to perform the actual tasks of the control software. If promotes loose coupling between components due to the application of the mediator pattern. Additionally the components are decoupled from the communication protocols using the communication abstraction framework, called Port. It permits the flexible exchange of the used communication libraries.

Microsoft Robotics Developer Studio

Microsoft RDS [53] is a service-oriented architecture for robot control software and simulation. It is based on the Concurrency and Coordination Runtime (CCR), which manages the asynchronous parallel execution of tasks and is implemented using the .NET framework. Additionally, it features visual programming tools and a 3D simulation environment. The exchange of information between multiple hosts is realized using the Simple Object Access Protocol (SOAP).

Miro

Miro [105] is an object-oriented middleware based on a layered application layout. For communication purposes it relies on the aforementioned CORBA implementation TAO. The software architecture follows the client-server model. It provides interface hierarchies, aiming to solve the conflict between generalization of sensor/actuator modalities and access to the special features of individual devices.

Orca

Orca [13, p. 231-251] is an open-source framework for developing component-based robotic systems focused on fostering the software reuse in robotic research and industry. The overall goal is to be widely applicable by imposing minimal design constraints. It requires the adherence to a set of predefined interfaces in order to preserve the interoperability with other components. Concrete implementations of drivers for a variety of robots and sensor hardware are provided in a subproject of Orca, called Hydro. While earlier releases were built on top of CORBA and SOAP, the current major version utilizes the general-purpose communication middleware Internet Communications Engine (ICE) [43] for inter-component communication. ICE provides cross-language and cross-platform communication mechanisms, which also includes remote method invocation.

Orocos

The Orocos project is organized in five libraries. The Real-Time Toolkit (RTT) provides a framework for developing real-time control systems. Therefore it also features a real-time communication layer, which is a rarely supported functionality. For components distributed across multiple hosts CORBA is utilized.

Two libraries of the Orocos project provide functionality for Bayesian filtering as well as modeling and computation of kinematic chains. The Components for Control library contains ready-to-use components for various applications. The latest library, the Simulink Toolbox, enables creating Orocos components in Matlab/Simulink.

Player

Player is a cross-platform robot device interface and server and due to its early release in the year 2000 widely used in robotics research and education. It makes no assumptions about the internal structure of the robot control software. But it provides a set of generic interfaces to a variety of sensor and actuator data. The interfaces follow a character device model. For communication across multiple processes or hosts the eXternal Data Representation (XDR) is utilized. Noteworthy is the limited usability for update rates higher than 100 Hz.

RoboFrame

RoboFrame [79, 81] is developed in the author's group and is object of this thesis. The software architecture is based on the framework paradigm. Due to the inherent application of inversion of control most interfaces of the software differ considerably from other approaches. It provides common features of a message-oriented middleware, but includes compile-time composition of components only, in contrast to the other presented component-based approaches. Within this work RoboFrame has been enhanced with multiple unique features as described in the following chapters.

Robotic Operating System (ROS)

The robot operating system [84] is a kind of meta-operating system initially developed at the Stanford Artificial Intelligence Laboratory. Besides the common features of message-oriented middleware it also provides uncommon system services like package management. It utilizes a custom messaging protocol, which is based on a central master for negotiating the message exchange. But the actual transfer of data is handled in a peer-to-peer manner. The focus is on reusability of the loosely coupled components and on sharing and collaboration. Since the development has moved to the company Willow Garage in 2008, it has been adopted by many researchers and is rapidly evolving. Various software repositories provide numerous components for different hardware and implemented algorithms.

Urbi

Urbi [2] is a software platform to control robots and is also open-source. It is based on a C++ component library called UObject, which defines a set of interfaces for common robot hardware and algorithms. Additionally, the script language Urbi script is used, which supports parallel and event-based programming paradigms. It is utilized to connect the components with each other and describe high-level behaviors. Urbi aims to simplify the process of programming robots and their behavior. Therefore, it brings sophisticated GUI tools along, which enable graphical editing of the behavior and a custom interface composed of parametrizable widgets. However, the visual robot control software Gostai Studio and Gostai Lab are proprietary and not freely available.

Other Approaches

Some approaches, which vary in the targeted hardware or aimed goals, are mentioned briefly in order to illustrate the inherent differences of the fields of application.

agile Robot Development (aRD)

aRD [46] from the Institute of Robotics and Mechatronics at the German Aerospace Center (DLR) is used to control the DLR Two-Hand-Arm-Torso, which has 43 degrees of freedom. The control rate is between one and ten kHz with hard real-time constraints needed for mechatronical systems. The custom message-oriented architecture is running on a cluster of powerful computers. The software features various different GUIs for debugging and monitoring the different aspects of the robot.

Aseba

Aseba [71] is an event-based architecture for distributed control of mobile robots developed at the Miniature Mobile Robots group at the Swiss Federal Institute of Technology in Lausanne. It targets integrated tiny robots equipped with multiple micro-controllers and utilizes a scripting language for programming the agents.

Carmen

Carmen [76] is an open-source collection of software for mobile robot control developed at the Carnegie Mellon University. While it provides functionality for inter-process communication, its focus is on reducing the barrier for implementing new navigation algorithms. It provides basic navigation primitives including obstacle avoidance, self localization, path planning and mapping.

2.4.5 Comparison and Evaluation

Even the limited set of presented existing solutions exhibits the various different aspects middleware is focusing on. Hence, the joint usage of multiple approaches for a single application is reasonable for many use cases in order to reuse already existing software. For example, the design concept of MARIE leads the way of integrating various existing solutions.

For a long time, interoperability between distinct projects was rare. Only recently several projects have approached one another. Orocos RTT can be seamlessly integrated into ROS applications in order to complement real-time features, which would otherwise not be supported. The same step has been made for Urbi, which can alternatively use the communication infrastructure of ROS and also be integrated with Player.

ROS is gaining significant importance due to the interoperability with various other projects. This represents a unique characteristic among the different projects.

Therefore, in this thesis the interoperability of RoboFrame with ROS is realized in order to take advantage of the complementary features.

But as long as the robotic domain *lacks a standard for interoperability*, the suitability of the different middleware must be individually evaluated for each specific scenario. It depends on the concrete requirements of the hardware and the scenario and the provided support for these demands from the middleware and existing components built on top of them. However, the aspect of interoperability with other systems is an important factor in the decision in order to develop future-proof applications.

2.5 Discussion

All middleware presented in Section 2.4.4 provides communication mechanisms to exchange data between the different modules of the robot control software. The thereby achieved decoupling of components enables a flexible composition of the components. Several approaches utilize general-purpose middleware like CORBA or alternative implementations of that standard. These solutions are not tailored for robotic systems and are designed with a focus on large scale distribution.

The communication between components running on the same host is handled similar as for distributed components by almost all middleware. Some implementations avoid the overhead of the IP stack for local communication, but still involve resource intensive marshaling, memory copy and demarshaling of exchanged information. Other approaches try to reduce the overhead

imposed by the middleware [66] but still pass the locally exchanged data through the IP stack. But in the context of mobile robots with quite restricted computational resources even this reduced overhead is unfavorable as discussed in detail in Section 6.2.

Equally, for most middleware, it is not possible to dynamically reduce the frequency for sending information to a remote monitoring station, without the need to manually reconfigure the data flow. Due to the limited available bandwidth this is a reasonable feature, when the communication layer is utilized for mobile robots.

For the tasks of debugging and monitoring the aforementioned middleware offer separate user interfaces specific for the respective algorithms. While applicable to analyze the particular parts of the control software it does not yield an adequate usability for debugging complex overall applications.

For scenarios with multiple cooperating robots the support of the presented middleware is sparse. While the explicit communication between multiple robots is supported by several of the existing approaches, the support for debugging and monitoring of teams of robots is non-standard. The presented applications for offline analysis of teams of collaborating robots in Section 2.3.7 demonstrate the complexity of the tasks. Neither of the mentioned middleware provide comprehensive tools or solutions built on top to address those specific demands thoroughly.

3 Proposed Methodology for Efficient Middleware

In this chapter several typical use cases from different applications in the context of autonomous mobile robots are identified and described (Section 3.1). All of them are related to efficiency, either in the sense of runtime efficiency or regarding the programming and usability of appropriate debugging abilities. Therefrom a set of functionalities and requirements for middleware is derived.

In Section 3.2 a methodology for evaluating the improved runtime efficiency of the middleware is presented. For aspects more difficult to quantify, the fitness of the approach is estimated using metrics from the domain of human-computer-interaction (HCI), which are applied for a set of common work cycles.

The chapter closes with Section 3.3 presenting the fundamental ideas and developed design concepts to improve the various aspects of efficiency.

3.1 Use Cases and Their Requirements

The requirements for robot middleware differ depending on the used hardware, the environment and the tasks the robot should carry out. In this section several use cases involving autonomous mobile robots are presented, which are typical for the considered scenarios and likewise are representative. Based on these use cases, the requirements for the middleware are constituted.

In the robotics domain hardware as well as software is evolving rapidly. The innovation cycles for newly developed sensors and actuators are short and the algorithms of the various different domains involved are rapidly developing.

Adaptability

For a component-based system integration the task of replacing single components is a common requirement. Either an existing component is exchanged with an alternative implementation of the same algorithm or a totally complementary approach is used to achieve the same goal. For simulation and testing this procedure is utilized to replace components interfacing the hardware with stubs, which connect to a simulator [36]. As the message-oriented communication layer provides the required decoupling of components these demands can easily be achieved by any of the above-mentioned middleware.

Another common use case is the modification of the application layout in order to adapt it to changed scenarios and tasks. This involves adding and removing components, changing the interface of components, which is equivalent to altering the message format, or modifying the flow of information between the components. For example, when testing the robot control software in simulation, ground-truth data can be provided by the simulation model [34] to replace possibly inaccurate computed information. Such modifications must be transparent to the components wherever feasible in order to achieve an improved maintainability. Even though all of the middleware can also handle such demands sufficiently, the required efforts vary for each of them.

Flexibility

A use case encountered when working with multiple robots, is to adapt the flow of messages dynamically during runtime. This is often needed when using the debugging and monitoring tools with multiple robots concurrently. Depending on the information of interest the message flow must by dynamically adapted to suit the current task. Some of the described middleware can modify the application layout and message flow during runtime, while other systems require a rebuild of the modified application, which might not be flexible enough to deal with such dynamic requirements.

Reusability

A further common application is the reuse of existing components either multiple times in the same application or in multiple different scenarios. The first example arises when, e.g., the existing image acquisition component is reused in order to build a stereo vision system. The second variant is even more clear when considering the two presented scenarios, which share multiple components for common functionality.

An issue, when reusing existing components, is the appropriate configuration for each concrete application. Therefore, the middleware must provide mechanisms to configure the component for their specific usage and especially the context of interacting with other components.

3.1.1 Restricted Onboard Resources

The usage of a middleware for wiring several components together does undeniably *reduce the dependency and coupling* between different parts of the software. It emphasizes the separation of concerns and therefore fosters the development of numerous small components, each focused on a single specific task. This is obviously an advantage from the software engineering point of view.

But in the scenario of mobile robots the payload and therefore the available computational power is commonly limited. Due to the restricted resources the runtime efficiency of mobile robot applications is crucial.

Efficient Local Communication

The additional layer of abstraction when using a middleware comes with the trade-off of an added overhead for passing information between components. Attention must be paid to the resources required for exchanging data, which usually involves *object marshaling, memory copy and object demarshaling* (Figure 3.1).

Therefore, the middleware must perform the message exchange as efficiently as possible. But the requirements for minimum resource consumption must not oppose the reasonable software design criteria and sacrifice the advantages of the decoupling.

3.1.2 Debugging and Monitoring

Due to the complexity of the applications, tools for debugging and monitoring are crucial for the development process. It is assumed that for every component of the control software at least

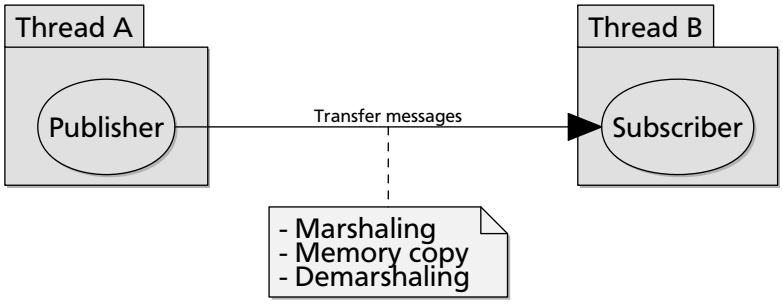


Figure 3.1: Overhead of message exchange due to marshaling, memory copy and demarshaling

one corresponding tool is reasonable, if not essential, for being able to debug and monitor them thoroughly.

The requirements for these tools depend on the performed tasks and the users of the software. This thesis focuses on users, who are developing, testing and monitoring robot applications and are adequately skilled in using different kinds of user interfaces.

Graphical User Interface

For many use cases a graphical user interface (GUI) is mandatory to provide complex visualizations of information like camera images and maps of the environment. Furthermore, a graphical representation of information, e.g., sensor readings, is usually better perceivable than a textual variant.

Due to the large set of different components in a complex control software multiple tools are often needed simultaneously. Therefore, these tools are comparable to integrated development environments (IDE) for pure software programmers. The requirement for the middleware is to provide the infrastructure for building an integrated graphical user interface, which is extensible to custom needs. It must be efficient to use in order to reduce the time spent for performing debugging and analysis tasks.

The same criteria as for the development of components for the robot control software also apply for the software design of the GUI. Decoupling between different tools as well as fostering flexibility, adaptability and reuse is again an eminent goal.

3.1.3 Restricted Bandwidth

A common use case in robotics is monitoring intrinsic information of the robot. Due to the mobility of the robots the tools for debugging and monitoring the internal state of the application are executed on remote computers. The bandwidth between the robot and the remote computer is often restricted to a wireless connection and therefore the amount of exchanged data is limited.

When visualizing information from the robot in real-time the amount of data required to transfer can overload the available bandwidth easily (Figure 3.2). An exemplary use case is to view the images of a robots camera at a remote computer to evaluate the quality of the perception. With an approximate size of half a megabyte per image and a frame rate of only ten frames per second any common wireless LAN connection would certainly be overloaded with the described showcase, since the 802.11g protocol provides only a maximum gross bandwidth of 54 Mbit/s which equates to less than four Mbyte/s (assuming an optimistic quota of 50 % net).

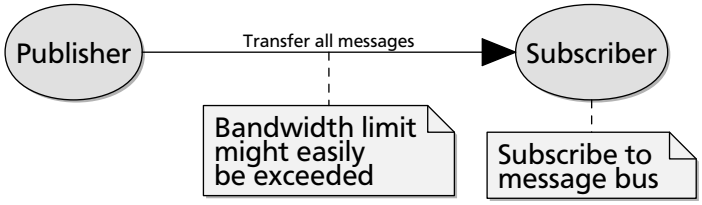


Figure 3.2: The subscription to a message bus implies transferring all messages. Without any kind of filtering the available bandwidth can easily be exceeded

Reduce Amount of Data

Reducing the amount of data is especially valid for messages exchanged at a high frequency or with a large payload size, e.g., raw sensor data. Therefore, the amount of information involved in debugging and monitoring needs to be reduced to fit the limited bandwidth available. This filtering must be carried out on the same machine where the messages are published and only transfer a reduced amount of data to the subscriber. However, in topic-based middleware the subscription is generally to a simple binary condition. It is not intended to subscribe to only a subset of the published information on a single message bus.

3.1.4 Offline Analysis

While independent functionality can be easily tested and debugged, the complexity of the overall system and the amount of involved data in autonomous robots makes it rather difficult if not impracticable to monitor in real-time. In many cases the frequency of changes is too high to allow a human to keep track of all relevant information.

The only alternative to online debugging and analysis is to conduct the task with recorded data offline. To make this feasible it is necessary to provide the capability of recording the massive amount of intrinsic data directly on the robot as the connectivity to an external system is usually limited in terms of available bandwidth. In a message based middleware recording the messages exchanged between the components is an easy task due to the design of the architecture itself.

Afterwards all available information can be inspected in detail by replaying the messages. Depending on the application this can be done in real-time, slow-motion or even frame-by-frame or message-by-message. Thus, the middleware must provide tools for recording and playing back any kind of message exchanged in the robot control software.

In order to exploit the full potential the recorded messages can be used for more than just introspection and visualization of the message content as described next.

Playback and Feed into Control Software

The recorded messages can also be replayed and fed into a customized control software. I.e. instead of using sensor data from the hardware, the recorded messages of the sensor data is used instead (Figure 3.3). Due to the offline characteristic it becomes much simpler to apply debugging techniques, which are unfeasible when running on the robot. The same logged messages can be replayed over and over again until an issue in one of the components has been identified, corrected and validated to be fixed. This approach is feasible for any component in the control software

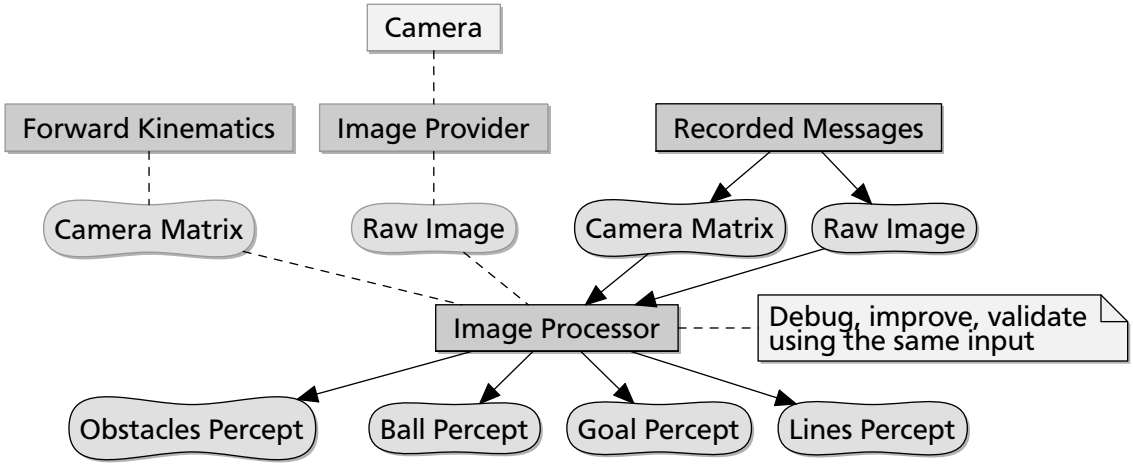


Figure 3.3: Recorded messages are fed to the control software bypassing several components

as long as the middleware provides a convenient way to alter the message flow according to the particular test case.

The amount of data may even be too voluminous to be archived on the mobile platform, since the available storage capacity is limited as well as the speed of writing to the storage device. Since an external storage is also impracticable due to the restricted bandwidth, it is necessary to limit the recorded messages to a subset. The subset may either skip specific messages entirely or reduce the frequency of messages if appropriate. For example, it is not necessary to record all raw images, when the debugging task is not focused on the image processing. However, the decision on a subset of messages is always a trade-off between storage size and completeness.

The desired reduction of messages corresponds to the previously described concept of filtering the messages for the remote monitoring task (Section 3.1.3). But in this use case the network bandwidth is not the limiting factor, as both endpoints are running on the same host. Instead, another overhead becomes apparent. In order to record some of the messages, the recording component must still subscribe to every relevant message bus. Since only a fraction of the messages may actually be stored, the overhead for marshaling and copying messages, which are not to be recorded anyway, takes up a certain amount of valuable CPU and memory resources (Figure 3.4). This overhead makes filtering the messages, just before recording them, still undesirable.

3.1.5 Teams of Robots

One of the considered scenarios involves teams of autonomous mobile robots. In this setting multiple robots have to act in collaboration to achieve the goals of the scenarios. Collaboration can be based on *implicit* as well as *explicit communication* [28]. Only the latter is considered in

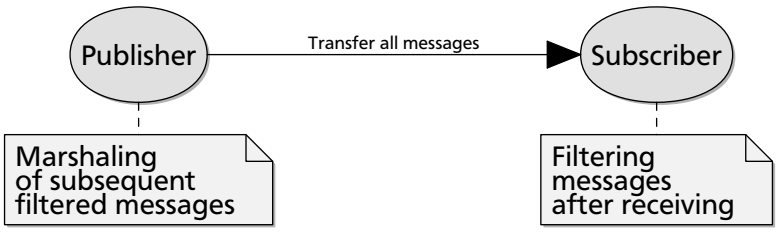


Figure 3.4: Overhead when messages are in the first place skipped by the subscriber

the following, because implicit communication cannot be supported by the middleware and is left to the application specific components.

Explicit Communication on Different Levels

The exchange of information with teammates can be applied at various different levels and elements of the control software. The most common use is sharing significant information of the world model between the robots. This may, e.g., include the estimated position, orientation and state of a robot as well as information about its environment like perceived objects in the surrounding area. Another element where team communication is also advantageous is the behavior control. For example XABSL provides features to ensure that only a specific number of robots stays in a particular behavior state simultaneously [86], e.g., a single robot approaching the ball.

Unreliable Network Communication

The communication between mobile robots is usually based on a wireless network. Depending on the environment the network can be unreliable and vary in the available bandwidth. In particular the wireless conditions during RoboCup competitions are continually congested, resulting in an environment with extreme conditions. Thus, each robot's application must be able to operate independently of the other robots in the team in case of connectivity problems.

Some of the above-mentioned middleware rely on a central instance to control the message exchange between the components on a single robot. This paradigm is not transferable to a team of mobile robots as the network connectivity cannot be guaranteed. Hence any concept with a central master involved in a team of mobile robots is impracticable for these scenarios. Still the concept of exchanging messages between multiple robots without a central master is adequate.

The middleware must therefore provide a mechanism for explicit communication between multiple agents, which run separate instances of the middleware. Depending on the scenario the exchange of information can utilize different communication paradigms like unicast, broadcast, or multicast.

Extending GUI to Teams of Robots

To monitor multiple robots simultaneously the graphical user interface needs to be able to communicate with multiple agents at the same time. But, due to the message bus based communication, existing GUI components designed for single robot communication are not able to differentiate between multiple sources.

In some use case this is not relevant as selected tools are able to visualize information from multiple sources simultaneously like, e.g., the position of multiple agents can be easily visualized simultaneously on a known map.

But in most use cases the tools can only deal with one robot at a time. An example is an image viewer showing the raw data of a robot's camera. When multiple agents provide their sensor data simultaneously on the same message bus the output is unusable due to the competing visualizations of images from many sources. Instead it is more desirable to use multiple instances of the same image viewer, but each tied to communicate with a single robot only.

The usage of separate GUI instances for each single robot does not fulfill the usability demands. When monitoring a team of cooperating robots in a combined view spanning features like, e.g., globally available keyboard shortcuts can be used in order to make the interface more efficient to use. This cannot be achieved when multiple separate tools would be used.

The same functionality as for online usage must be provided for later offline analysis as well, when only using the recorded logfiles from each robot. This use case makes high demands on the features of the used middleware. First, it must be able to distinguish the source of the recorded messages. Second, it must allow to playback these information as if they are virtually coming from different robots. This requires complex dynamic message passing and routing capabilities in order to not depend on the user to conduct numerous configuration changes manually.

3.2 Evaluation

The impact of the proposed and implemented concepts must be objectively quantified. For the aspect of runtime efficiency of the communication layer common benchmarks are used to measure metrics like scalability, throughput and latency. The evaluation of the debugging and monitoring tools for usability criteria and efficiency of usage is more challenging.

3.2.1 Benchmarking Runtime Efficiency

For evaluating the performance of a message-oriented middleware benchmarks are applied. They can be used to compare different applications as well as to measure the performance before and after the modification of a single application in order to determine the impact of the changes. But developing a good benchmark is a challenging task as described in [49].

For example, a popular benchmark for enterprise MOM based on JMS is the SPECjms2007 from the Standard Performance Evaluation Corporation (SPEC). It addresses two distinct topologies:

- *Horizontal Topology* — the number of destinations is increased, while keeping the traffic per destination fixed.
- *Vertical Topology* — the traffic per destination is increased , while the number of destinations is constant.

These measurements are focused on quantifying the scalability and throughput of enterprise middleware. But for the considered scenarios these criteria are neither important nor a limiting factor. More relevant is the latency, when passing a single message with a specific size from a subscriber to a publisher as documented later in Section 6.2. Therefore, the SPECjms2007 is not suitable for the targeted scenarios.

Nevertheless, conducted benchmarks with MOM from other domains can provide a rough magnitude of expected latencies for message passing. In [88] the average latency of a message is in the order of milliseconds. On the one hand, these tests have been executed on powerful server hardware in contrast to the restricted computational power in the considered scenarios. On the other hand, the endpoints were running on separate machines, which induces an additional latency due to the network involved.

Commonly the endpoints are distributed across multiple machines. But in the considered scenarios the focus is on endpoints running in the same process or even in the same thread.

Measuring Latency

The relevant latency for point-to-point communication can be measured easily. Therefore, a message is sent from a publisher to a subscriber and the timestamps when sending and receiving the message are compared (Figure 3.5). In order to yield a repeatable result, the measurements of numerous messages need to be averaged. Other effects relevant for the benchmarks require warm-up and cool-down phases in order to achieve fair results. A comprehensive description of the various relevant aspects on benchmarking in general can be found in [54].

However, this approach has multiple drawbacks:

- *Short Interval* — determine the differences between the two timestamps requires a high accuracy of the measured time, especially when both endpoints are running on the same host.

- *Multiple Messages* — the publisher is not aware when the message has been received by the subscriber. Therefore, it is unknown when to publish the next message. Either the next message is sent too early, which results in a higher measured latency, or it is sent later than necessary, which increases the duration of the benchmark needlessly.

- *Distribution* — when the endpoints are running on different hosts comparing the timestamps becomes a challenging task. Synchronizing the time between the hosts is often not accurate enough and does not account for the existence of time drift during a test run.

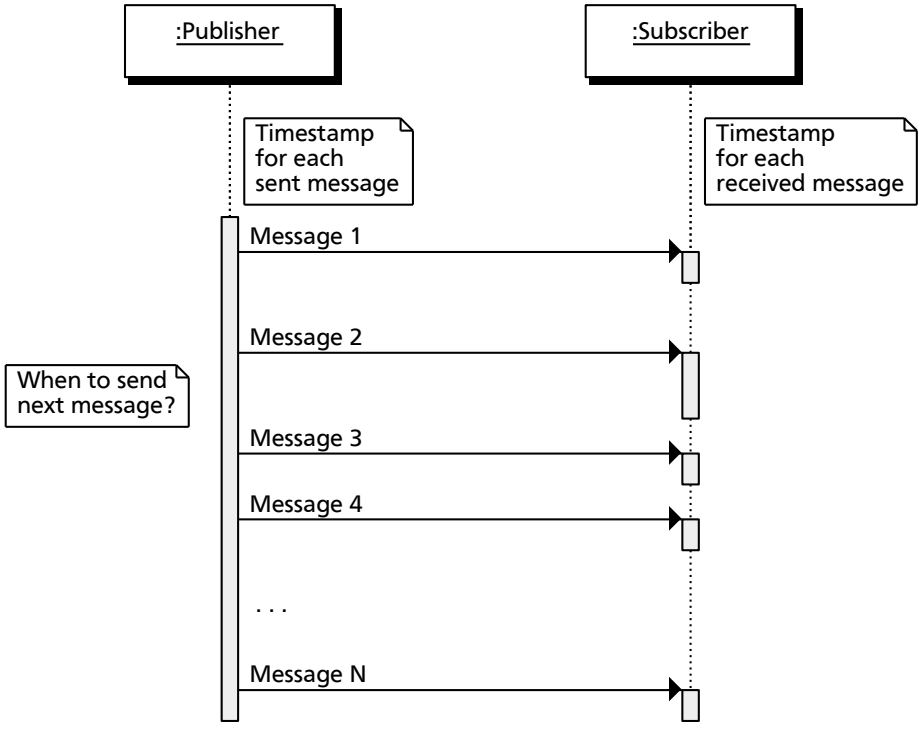


Figure 3.5: Measuring latency of message exchange using unidirectional communication

Round Trip

Hence, commonly a different approach is chosen to measure the average latency. Instead of sending messages unidirectional, which requires timestamps on both endpoints, the messages are passed round trip (Figure 3.6).

This circumvents all disadvantages of the above-mentioned approach:

- *Less timestamps* — for an average latency only the timestamp of the first message sent and the last message received is required. Optionally the latency of every single round trip could be measured, in order to generate detailed statistics including evaluations like the standard deviation.

- *Sequential round trips* — the publisher sends the next message after receiving the response of the subscriber to the previous message.

- *Timestamps on single host* — as the timestamps are only taken on a single host, the time offset as well as the drift are irrelevant.

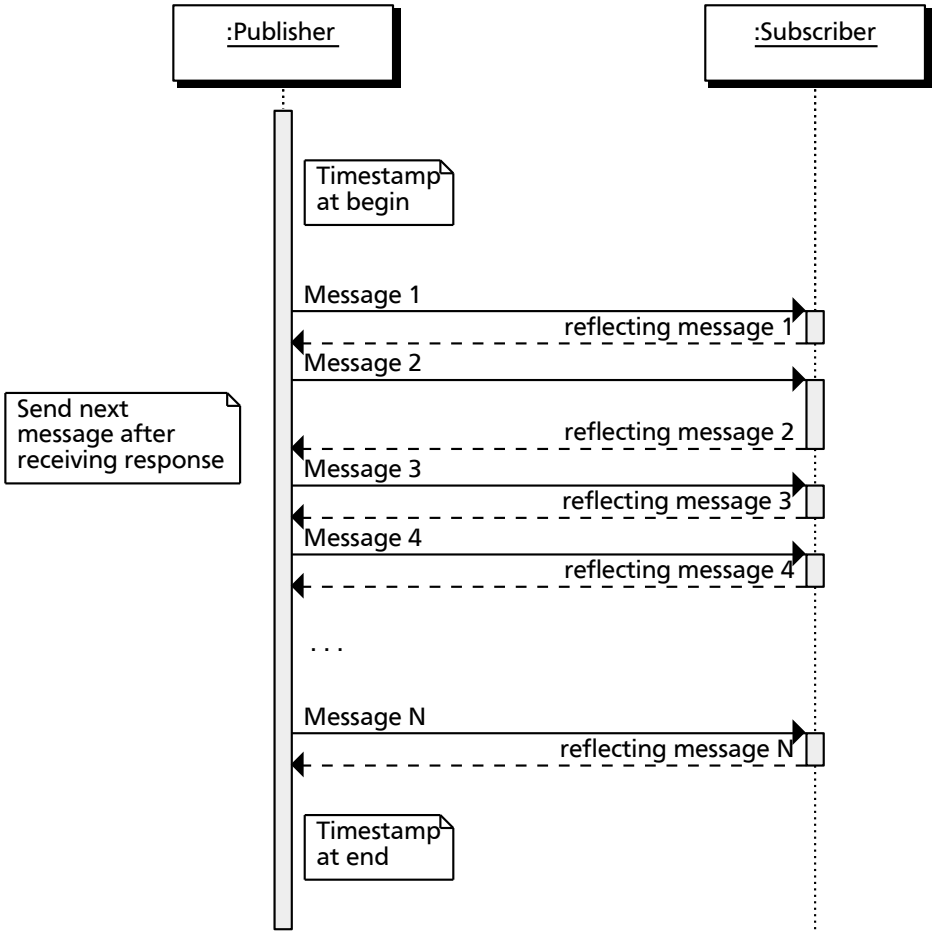


Figure 3.6: Measuring latency of message exchange using round trip communication

Different Kind of Messages

Furthermore, the type of message has a significant impact on the results. The messages vary in the following aspects:

- *Message Size* — the payload of a message may range from single bytes representing states to megabytes of data for readings from sensors like a camera.
- *Type of payload* — a message could either contain raw byte data, which do not need any marshaling, or consist of complex nested objects, which require additional marshaling and demarshaling operations.

Therefore, the benchmarks utilize various kinds of messages in order to determine how the results vary for different amounts of data and the type of payload.

3.2.2 Quantifying Usability and Efficiency of Debugging and Monitoring Tools

Benchmarking aspects like usability and efficiency of use are catchier than measuring pure runtime performance. In the domain of human-computer-interaction (HCI) different models of the man-machine-interaction have been developed. The best known is *GOMS*. The acronym stands for *Goals*, *Operators*, *Methods* and *Selection rules*. It is a family of modeling techniques that analyzes the human-computer interface interaction and is presented in [16].

GOMS reduces the interaction of the user with a computer to elementary actions, which can be either physical, cognitive, or perceptual, and estimates the efforts related to each of these. It is commonly used to predict the time a user will need to carry out a goal. Multiple different variants of GOMS have been developed.

3.2.3 Keystroke-Level Model

A simplified and easy to apply variant of GOMS is the *Keystroke-Level Model*, sometimes referred to as KLM-GOMS. The application of the model is explained in [60]. For each user operation the duration is defined, which a user would take to perform in average. A subset of the different operations and assigned durations is depicted in Table 3.1.

3.2.4 Defining the Scenarios

In the following several scenarios are described, which cover distinct aspects when debugging and monitoring complex robot control software. By these exemplary tasks the reduced efforts necessary to achieve the goals are recorded in later chapters.

Prepare Multiple Tools for Monitoring Task

In order to perform any kind of monitoring task the required tools must be prepared. Initially each single tool must be started and arranged clearly. Additionally, the tools must be configured to communicate with a specific robot in order to receive information, which has to be visualized. After that the actual monitoring task can be conducted.

Table 3.1: Subset of operations and assigned durations in the Keystroke-Level Model

Operation	Description	Duration [s]
K	Key press and release (keyboard)	0.08 - 1.20
	— Best typist	0.08
	— Good typist	0.12
	— Poor typist	0.28
	— Typing random letters	0.50
	— Typing complex codes	0.75
	— Unfamiliar with keyboards	1.20
P	Point the mouse to an object on screen	1.10
B	Button press or release (mouse)	0.10
H	Hand from keyboard to mouse or vice versa	0.40
M	Mental Preparation	1.20

When the same task is repeated later or another task is started the same sequence of actions needs to be reiterated.

Analyze Data from a Team of Robots Offline

To analyze data from a team of robots offline, all relevant messages need to be recorded in advance. Again every used tool must be started and arranged. Then the logged information from each robot must be loaded and somehow connected to the respective visualization.

Then the user can navigate through the large amount of information, which occurred during the test, to analyze specific situations. Once the corresponding timeframe has been located, the appropriate data can be examined using the various specific tools.

Review Data for Known Issues

Another, often pursued, goal is to review the recorded data for known issues. A common problem for mobile robots is to detect discontinuities in the self localization. Particularly interesting for biped walking robots are occurrences when a robot fell over.

As in the previous scenario the tools need to be prepared and fed with the recorded messages. Then the operator can browse the data pool to identify timeframes where these issues of interest occurred.

After a problem has been located the reason for it must be identified manually using the available information.

Compare Different Parametrizations or Algorithms

The last example of considered goals is about comparing different algorithms or different parametrizations of the same implementation. This is a very common use case in robotics, since many algorithms support numerous parameters and various different approaches exist in each subdomain.

Both variants are processing the same input data. The results can then be compared and a decision can be made, which one performs better and perhaps quantifies the disparity.

3.3 Concepts for Robotic Middleware Investigated in This Thesis

In the following, the concrete ideas and concepts for efficient programming and middleware design are described in order to address the specific requirements of autonomous mobile robots and complex software design. Several concepts adhere to the concept of a framework rather than that of a library (Section 2.2.4) which differs significantly from other middleware implementations.

3.3.1 Efficient Local Message Exchange for Common Application Layout

The messaging performance of the aforementioned middleware is fairly homogenous as compared in Section 6.2. The introduced overhead is relatively low and in common applications neither a significant nor a limiting factor. But nevertheless, depending on the scenario and the application, the overhead introduced by the message passing concept becomes significant. Especially for mobile robots with very restricted hardware resources, any possibility to reduce the consumed resources is considered reasonable as mentioned in Section 3.1.1.

Middleware is designed for scenarios where decoupled components are executed concurrently in separate threads, different processes, or even distributed across multiple machines. This makes copying messages imperative. But, depending on the application, it is often reasonable to *execute* selected components on the *same host* as well as *sequentially*. One common use case in robot application, which fits the approach of serially executing components very well, is the sensor data acquisition and processing.

Generally, it is feasible to execute such components consecutively in a single thread as their execution order is semantically tightly coupled anyway. This approach implies that the two operations of acquisition and processing must be executed in succession. It does not require the processing to be performed single threaded; it could still be partitioned into multiple worker threads.

Still, the separation in two distinct components is valid as it allows the flexible exchange of each single implementation, e.g., replacing the sensor acquisition component which interfaces the hardware with an equivalent connecting with a simulator.

Combine Components in a Single Thread

The main advantage of combining components in a single thread is that it permits *passing references* (Figure 3.7) for messages between them without any kind of overhead or need for mutual exclusion. Marshaling and memory copy is not required for such intra-thread communication. The described use case of sensor data acquisition and processing is especially suitable in two different ways. It is present in any robot application and therefore relevant for any scenario. In particular, the raw sensor readings are usually one of the largest data blocks, which are exchanged between components. Thus, reducing the overhead for this kind of exchanged information proposes large gains in performance.

If other external components, not combined in the same thread, are also subscribed for these data, they need to be marshaled and copied anyway.

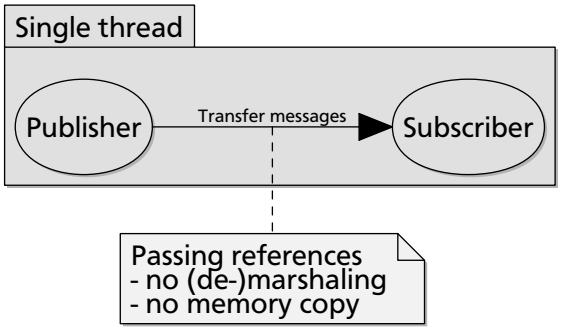


Figure 3.7: Avoiding overhead of message exchange due to passing references

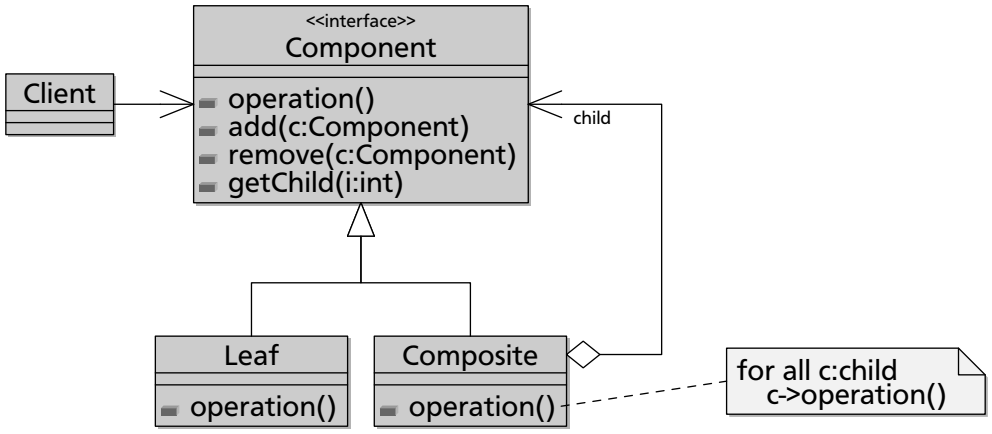


Figure 3.8: The Composite pattern

The only solution providing a similar kind of feature as developed in this thesis is ROS (Section 2.4.4), which introduced a new concept called *Nodelets* in the latest release[1]. Thereby components implementing the ROS nodelet interface can exchange messages using *Boost* [57] *shared pointers* with other nodelets executed in the same runtime environment. More information on this new approach can be found in the ROS wiki[2].

Composite Pattern

Following the *Composite* pattern as depicted in Figure 3.8 [39, p. 163-173] a meta component is used to aggregate multiple components. The described approach of composing multiple components in a single meta component must be transparent for both the components and the middleware, since at programming time it is not yet known if the component will be deployed distributed, on the same host or even be composed together with other components. The middleware must only be aware of the composite component, but not of the nested subcomponents. Likewise, the subcomponents are not aware that they have been composed by the composite.

For the specific goal of improving the message passing performance between the subcomponents, the meta component can relay messages published by one subcomponent to any other subcomponent directly by passing references. This avoids any kind of overhead normally introduced by the middleware.

[1] ROS Version 1.2.0 released in August 2010
[2] http://www.ros.org/wiki/nodelet

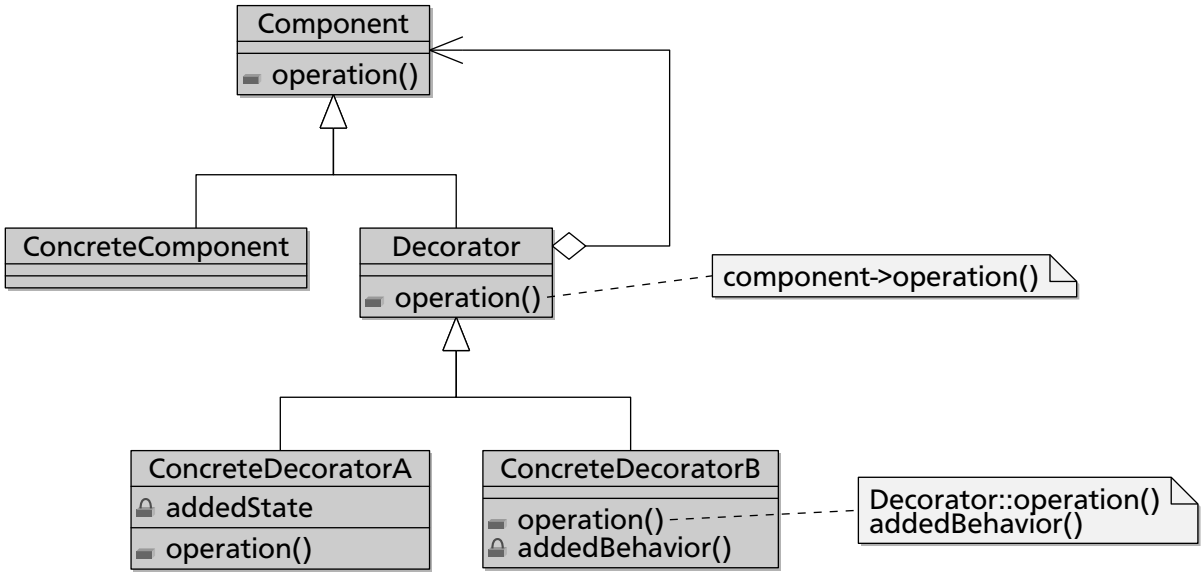


Figure 3.9: The Decorator pattern

In order to be able to alter the behavior of the interaction of the components with the middleware and optimize the message exchange between subcomponents as proposed, the meta component must be aware of the publications and subscriptions of all subcomponents. Furthermore, the composite must be responsible for passing the incoming and outgoing messages, so that it is able to manipulate any interaction between the subcomponents and the middleware.

The composite pattern also can easily be nested, whereas a leaf can again act as a composite. As a result, it becomes possible to apply multiple different behaviors one after another. This variant complies to the *Decorator* pattern [39, p. 175-184], which is depicted in Figure 3.9.

In order to apply a custom behavior it is necessary to *hook into the interaction* between the components and the middleware. But for the majority of existing middleware the communication layer is used as a library from within the component. This makes changing the interaction from the outside impossible (Figure 3.10).

Further concepts, described in the following, also require the capability to apply custom behaviors for the interaction. Hence, in Section 3.3.6 an approach is presented, which provides the necessary features to address these requirements in a flexible way.

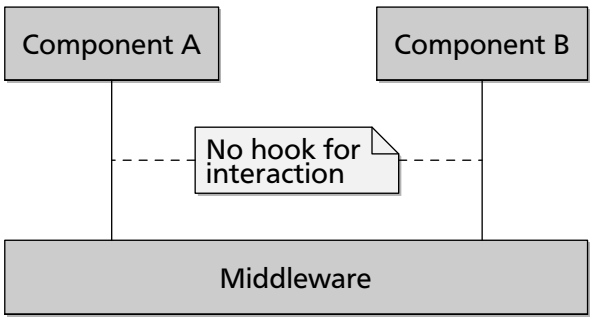


Figure 3.10: Missing flexibility points for altering the interaction between the components and the middleware

3.3.2 Integrated GUI

The graphical user interface needs to address many different use cases of debugging and monitoring and their requirements (Section 3.1.2). The various tasks involve different interfaces and visualizations to be supportive in reaching the desired goal of monitoring intrinsic information of the robot. While several separate tools certainly address the demands, they fall short regarding efficient usability and convenience features.

For example a developer is used to work with IDEs providing comprehensive features like customizing the interface with different views and toolbars for specific needs. Additionally, these configurations can be saved and restored to easily switch between different perspectives. Furthermore, in an integrated interface, global shortcuts for faster access to common actions are possible.

In principle an integrated GUI can provide numerous advantages compared to individual separated tools, which increase the efficiency of working with these tools. It permits the best type of interface to help the developer to reach the specific goal at that time.

Extensibility with Custom Elements

A GUI provided by the middleware can provide the infrastructure for an extensible interface and a set of common but application independent elements. To meet the application specific demands the GUI needs to be extensible with custom elements, e.g., a widget for displaying the images from a camera.

These elements are similar to the components of the robot control software. They need to be able to exchange information with each other as well as with the application running on the robot and therefore utilize the same communication layer. Additionally, the aforementioned software quality criteria apply, e.g., decoupling, flexibility and reusability. But in contrast to the components running in the control software, these components feature a graphical interface.

The infrastructure must not constrain the types of graphical elements that are usable. Anything ranging from windows, dialogs and wizards over toolbars, single actions and global keyboard shortcuts is reasonable depending on the use case.

Even non-graphical elements are conceivable. For example, developers are used to a command line interface. It provides a more efficient interaction for a subset of tasks and is therefore considered as an auxiliary interface for interactions.

Generic Features and Functionality

The infrastructure can provide overall features like opening and closing custom graphical components dynamically, saving their state on close and restoring it after a restart as well as switching between multiple perspectives as known from common IDEs. Additionally, implementations of common graphical components, which are applicable for the majority of applications, can be supplied as well.

A generic feature provides control for connecting with the applications running on the robots to exchange information. Other applications are widgets, which display the console messages of a remotely connected robot or introspect any received message.

3.3.3 Moving Filtering Data to Publisher-Side

The next concept is designed to reduce the overhead in utilized bandwidth as mentioned in Section 3.1.3. Therefore, the number of messages must be reduced before being sent by the middleware.

As the procedure of filtering must be separated from the subscribing component the configuration of the filter must be, unavoidably, passed from the subscriber to the entity, which carries out the actual reduction.

One possibility is to introduce an additional component, which is placed between the publisher and subscriber (Figure 3.11). It carries out the actual filtering operation and relays the remaining messages to the subscriber. This approach corresponds to the *Message Filter* pattern [48, p. 237-242], which is actually a special *Message Router* [48, p. 78-84] with a single output channel. Thus, the bandwidth used between the introduced component and the receiver is reduced to the essential amount. As long as the publisher and the filtering component are running on the same host, the still inflated bandwidth usage between them is of less importance.

While such an approach is easily realizable with existing tools, it still has two major drawbacks. First, it does not account for the overhead due to the still required marshaling and memory copy of later skipped messages between the subscriber and the filtering component. Second, the approach requires a high amount of modification of the application layout, especially, when applied to any exchanged data, and bloats the graph of message flow considerably.

Commonly a message filter is used to perform content-based filtering, which requires knowledge of the internal message structure. The functionality considered here, to only skip messages to reduce the quantity, is therefore named *throttling* in the following, to avoid any ambiguity.

Throttling Data on Publisher-Side

Opposed to subscribing for all information and then throttling the received messages on the subscriber side (or in an intermediate throttle component) the amount of exchanged data is directly reduced on the publisher side (Figure 3.12). This avoids any overhead for filtered messages.

On the one hand, the filtering operation must be transparent to the publishing component and is therefore performed after the data have been published. On the other hand, the data must be filtered before the middleware marshals the information and transfers the message to all subscribed receivers.

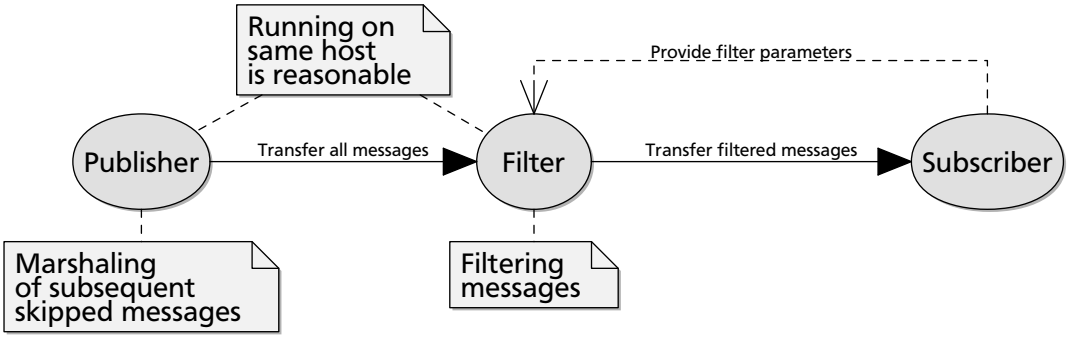


Figure 3.11: Reducing number of exchanged messages utilizing a message filter

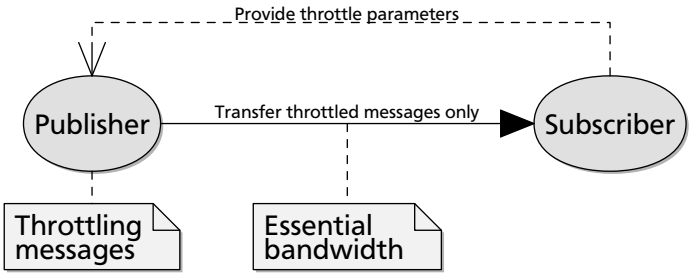


Figure 3.12: Throttling amount of messages directly at the publisher-side

Using the publish-subscribe messaging paradigm the publisher side must therefore be aware of the throttling parameters of each subscriber. In addition the throttling parameters of multiple receivers must be considered jointly. The messages are not always passed to all receivers but only to a subset, which must be supported by the underlying communication mechanisms.

If the subscriber specifies its throttle parameters during the subscription, they have to be passed to the publisher side. Based on this data a custom behavior of the interaction between the publisher and the middleware can perform the actual throttling even before the messages are handed over to the middleware.

Provide Meta Information to the Components

Besides using these parameters for throttling on the publisher side directly, another advantage is achieved in making these information available to the publishing components itself. A publishing component can skip complex calculations if neither of the subscribed components are requesting the next message due to the current throttle configuration (Figure 3.13). This enhances the efficiency of the overall application even further.

A common use case could be to subscribe to images from a remote robot application. Due to bandwidth limitations transmitting raw images is in general not efficient. Instead JPG compressed images are sent, which are sufficient for most use cases. In order to reduce the utilized bandwidth even further, JPG images are requested only once every second. Due to the knowledge of the throttle configuration the component, which performs the compression of the raw images, is able to skip the conversion for skipped data to additionally reduce the utilized CPU resources.

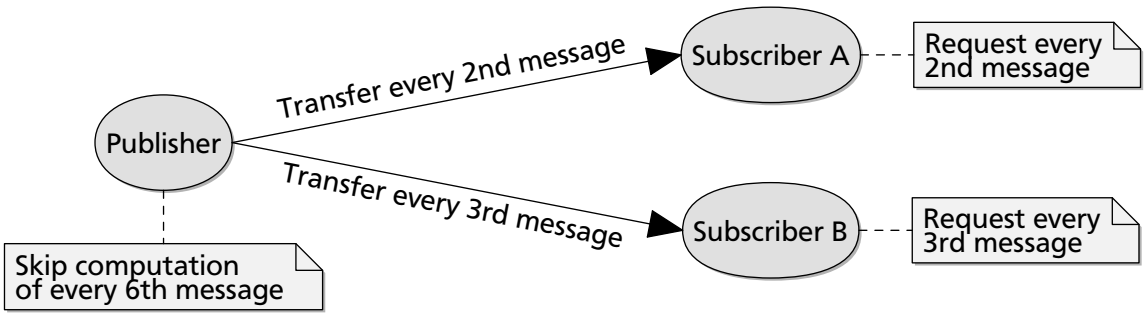


Figure 3.13: Provided meta information enable the publisher to skip unneeded computations

3.3.4 Offline Analysis

For analyzing information offline (Section 3.1.4), the available data must be comprehensive enough to obtain a detailed picture of the robot control software during the mission. Therefore, depending on the region of interest of the analysis, a different subset of the messages must be recorded at runtime of the robot's application.

The task of storing relevant messages is either executed on the robot or by using a remote tool like the GUI. The choice mainly depends on the amount of data collected, the available on-board storage capacity and the available bandwidth between the robot and the remote storage.

The subset of the messages to be archived needs to be highly configurable to suit the different demands depending on the use cases. For each kind of message a different throttle configuration is required, in order to record enough information to satisfy the analysis, but reduce the amount of exchanged and stored data to a reasonable size. The described scenarios provide a good estimation of how much data is exchanged inside an autonomous robots. A quantity of several hundreds of exchanged messages per second is not uncommon for complex applications as detailed later in Figure 6.3. The message data of the raw images alone accumulate to approximately one Gigabyte per minute.

The overhead introduced by the recording needs to be minimized in order to permit its usage during normal operation. Thus, the described concepts for improving the performance of the throttling of messages are crucial to permit such recording functionality.

Playback of Recorded Messages

The logfile containing all recorded messages can later be played back for detailed analysis.

Due to the quantity of messages being archived even in short periods of time, the task to examine the information becomes increasingly complex. The chronological data can be navigated using a time- or frame-based time line. But since the number of messages can easily become overwhelming, a different concept for navigating through this data is necessary to allow an efficient review of the available information.

To ease the process of navigating through the data, additional information must be provided to the user. This can be achieved through an automated extraction of prominent points in time, which support the user in browsing the information. These points in time are named *events* in the following. Depending on the scenario a custom algorithm detects specific situations or settings, which may be noteworthy. Several different algorithms can be applied simultaneously, each able to identify a specific condition, and generate labeled events at the specific point in time for better navigation.

For example, in the described scenario of soccer playing robots, the decisions of the referee are extracted and made available for navigating through such events.

Automated Analysis

Since the amount of data recorded for offline analysis is tremendous it becomes a challenging task to inspect all information in a reasonable time. The manual review process is time consuming and error-prone and since it is not very efficient it is often neglected. Therefore, the software architecture needs to provide a convenient way to automate the analysis and search for very

specific well known issues. Two different use cases from the RoboCup scenarios are exemplary discussed.

The first use case is taken from the domain of self localization. Under the assumption that a mobile platform is only moving in space in a continuous manner and with limited velocity, the position of the robot should always be within a reasonable distance from the previously determined location. If the position of the robot in the environment changes within a short period for a major distance, the conclusion is, that at least one of both locations, the previous or the current one, must be wrong.

The second use case is taken from the area of behavior control. A well know approach to implement behavior control is to utilize hierarchies of finite state machines. A common problem when working with state machines is the issue of oscillating states. Normally, this is due to wrongly specified transition conditions or insufficient hysteresis. Finding such situations manually is often very difficult, as the duration is commonly very short and often not perceivable when observing the robots' actions.

Instead of manually cycling through the voluminous amount of data an algorithm is used to detect such problematic situations. It already eases the manual review process, if the automatic analysis is used to indicate problematic situations in a well-arranged overview like the events described above. This information provides a good direction of where it is most efficient to manually look into a particular subdomain at a specific period in the recorded data.

External Reference Data

Compared to the online alternative, the approach of analyzing the information offline entails an additional problem. When reviewing the recorded data, the actual situation, e.g., the position of the robot, and the state of the environment is no longer visible. Therefore, it is difficult to evaluate the correctness of the data based solely on the intrinsic information, e.g., the perceived objects and the world models of the robot itself.

To provide sufficient information to allow an objective review of the intrinsic data, additional external information must be available as a reference. Any kind of external information can be used to provide an overview or detailed measurements of the environment. For a manual review, an external video camera provides a good starting point for humans to be able to assess the correctness of the intrinsic information and the behavior.

For ease of use it is required to be able to record additional external information and support a synchronized playback for the analysis. The synchronization can be performed automatically if a global clock is available or the messages are recorded on a central host. Otherwise they must be synchronized manually using specific points in time, which are determinable in both sets of the recorded data.

3.3.5 Extending to Teams of Robots

Team Communication

The requirement to exchange explicit information between teammates, in order to improve the cooperation in a team of robots, has been stated in Section 3.1.5. For an increased flexibility, the solution must be able to dynamically add additional robots to the team communication, which requires the middleware to adapt to these changes.

The team communication is used to share information of the world model with the teammates. Using this data, more complex behaviors can be achieved, e.g., a dynamic role assignment in order to improve the team behavior.

Besides the exchanged payload, meta information is also relevant in this use case. The middleware needs to provide some kind of identification of the source of received messages. This additional information is required by many algorithms for fusing the messages from teammates into the world models of each robot.

Network Topology and Transport Protocols

The initial choice between TCP and UDP is based on the characteristics of both protocols. As the team communication continuously sends up-to-date information to teammates, the decision was made in favor of the user datagram protocol, due to the lower latency, minimal overhead and availability of additional transmission types than unicasts. The decision on whether to use unicasts, multicasts or broadcast is explained in detail in Section 4.4 after the specific characteristics of the wireless network in the considered scenarios are presented.

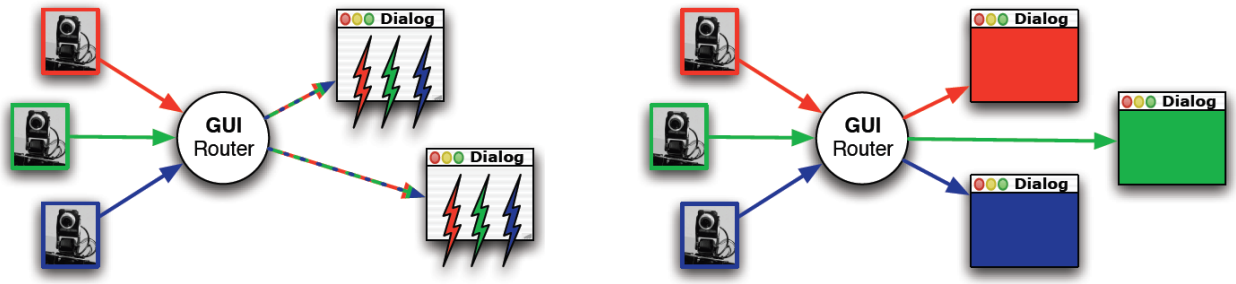
The communication in the team is not restricted to exchanging information between multiple real robots. The cooperative behavior is mainly developed and tested in a pure simulation environment without employing any hardware. This prevents the hardware from wearing and simplifies the execution of the tests. Depending on the complexity of the simulation and the test case, all robots can be simulated on a single machine. Sometimes the simulation of multiple robots is distributed across two or even more systems.

In all of these different scenarios, the robots of a team must be able to exchange information. These diverse use cases pose an additional challenge on the flexibility of the implementation.

An additional demand arises from the rules of the soccer playing robots. It forbids any kind of interaction with the robots, especially sending any data to them. It allows listening to the messages exchanged, thus the team communication is also used for monitoring the team during an ongoing game. Strictly speaking, TCP is not permitted due to its bidirectional conception of the communication.

Reusing an Existing GUI with Multiple Robots

In order to debug and monitor teams of robots, the GUI will be used to connect to multiple robots simultaneously. For some tools, like the indication of a robot's position on a map, this works immediately with multiple information received at the same time. But several elements of a GUI might not be able to handle messages received from multiple robots (Figure 3.14a). These need to be restricted to communicate only with a single robot at a time in order to work properly.



(a) The information from multiple robots cannot be visualized simultaneously

(b) Multiple instance of the same view, each tied to a single robot

Figure 3.14: Visualization of intrinsic data from multiple robots (Images adapted from [99])

However, multiple instances of the same GUI element can be used simultaneously to visualize the same kind of information, but from distinct robots (Figure 3.14b). These multiple instances can be considered an optimal reuse of existing functionality in the context of multiple robots. Again, a special kind of filtering needs to be applied to the messages.

One way is to make every GUI element aware of multiple robots and integrate the filtering in each single element. Apart from the undesired coupling to this feature, another disadvantage is, that each GUI element needs to provide controls in the UI for choosing the data source to use. However, this is necessary even if only a single robot is connected.

A superior solution is to utilize a central component, which provides a specific UI to configure the filtering of all active GUI elements. It is responsible to hook into the interaction between the middleware and the GUI elements and apply the filter configuration according to the current setting. The concrete GUI element is neither aware of multiple connected robots nor is the communication constrained between an external component and a single robot. Another benefit is, that as long as only a single robot is connected, the specific UI for configuring the filtering can be blanked out.

For GUI elements, which are capable of visualizing data from multiple robots simultaneously, the communication does not need to be restricted.

Offline Analysis of Teams of Robots

The extension to multiple robots reveals a common problem for the offline analysis. Since the collection of data is distributed across multiple hosts, it must be taken into account that the internal clocks can differ. Unfortunately, synchronizing the clocks is not feasible under all circumstances in the considered scenarios. Therefore, another approach is pursued.

Instead of avoiding any differences between the clocks on multiple hosts, it is sufficient if the differences are known. In the step of the offline analysis these known deltas can be regarded when playing back the messages from multiple sources. Ideally, the synchronization of the playback is done automatically without the need for any manual intervention.

First, the time offset between two hosts must be determined. This can easily be achieved during the normal operation by exchanging some timestamps between the hosts. The accuracy of the procedure is satisfactory for the described use cases, even when not being overly precise.

In order to make this information available for the offline analyses, it is recorded in the logfiles as well as any other messages used for debugging. When later multiple logfiles are loaded for playback they are searched for information about time differences. These measurements then can be regarded when playing back messages from multiple sources synchronously.

In order to enable the application of the filtering concept for each GUI element, as described above, the messages played back from the multiple logfiles must pretend to originate from distinct robots. Thus, for every logfile loaded, which comes from another robots, a virtual robot is created, which acts as a distinguishable source for the filtering. Both use cases, when connecting to multiple real robots and when loading logfiles from multiple robots, can then be treated equally.

3.3.6 Influence on Middleware Interface Design

The aforementioned use cases, the derived requirements and the developed concepts could be achieved in several different ways.

One approach is to implement the various different features on top of existing solutions. E.g., components, which need to exchange large amounts or frequent data, could easily be merged to use any custom mechanism for passing information internally. In the case of throttling, the receiver could, e.g., repeatedly unsubscribe and resubscribe to the message bus to reduce the number of transferred messages. In the domain of the graphical user interface, which supports communicating with a set of robots simultaneously, any user interface component could be adapted to work well with messages from multiple sources. But all of these options obviously suffer severe drawbacks regarding to the concept of separation of concerns in modular software design and the software quality criteria of low coupling.

Another approach is to integrate any of these features directly into the middleware. For the described concepts like composites and throttling at the publisher that would be sufficient. But it is very likely that new ideas and concepts will come up, which cannot be covered by the existing infrastructure. Certainly, a current solution cannot address any kind of feature in the future. But when the design permits customizing the interaction between the components and the middleware, it enables a wide range of custom solutions. The described concepts of the composite pattern for nesting components and the application of throttling directly at the publisher are only the "showcases" handled in this thesis. But in the future other ideas may evolve, which also rely on a custom functionality introduced between the middleware and the components.

Flexibility Point between Components and Middleware

Nearly all of the middleware presented in the previous chapter have one property in common. The components implemented for that middleware use the messaging functionality internally as a library. This tight coupling between the components and the API of the middleware limits the flexibility of the software significantly. Several of the aforementioned concepts, e.g., the application of the composite pattern, are impossible due to the way the components interact with the middleware. From the outside it is not possible to hook into the interaction between the components and the middleware mentioned in Section 3.3.1 and depicted in Figure 3.10.

It would be an ideal use case for the aspect-oriented programming paradigm [58] to allow hooking into this interaction. But such methods are only available in a small subset of programming languages.

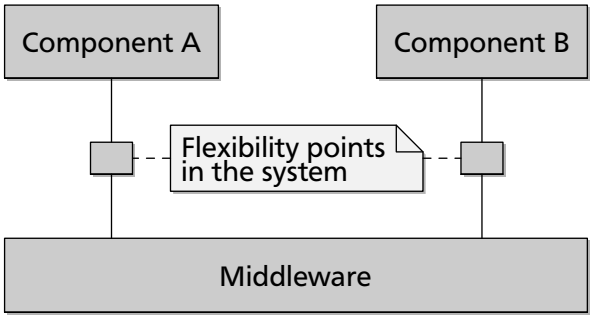


Figure 3.15: Flexibility points between the components and the middleware enable altering the interaction

Instead the middleware interface can be designed to support such hooks natively as illustrated in Figure 3.15. By using such hooks the behavior of the interaction between the components and the middleware can be altered arbitrarily.

Gateway Pattern

The flexibility point can be implemented using the *Gateway* pattern [30, p. 466-472] as depicted in Figure 3.16. The gateway is used by the components to encapsulate access to an external system, in this case the middleware. It reduces the coupling between the components and the middleware and comprises the minimal set of required interfaces to provide all necessary functionality of the service used.

In contrast to the *Adapter* [39, p. 139-150] and *Facade* [39, p. 185-193] patterns, a gateway does not necessarily alter or simplify the interface of the service used, but may just copy the wrapped interface entirely.

Theoretically, it would be possible to target multiple, homogeneously designed, middleware with different implementations of the same gateway interface. But designing a common interface, which fits different existing solutions, is extremely difficult and out of the scope of this thesis. More feasible is the approach, which is part of the ongoing BRICS project [6], to establish a set of uniform messages for simplified information exchange between existing technologies.

Besides the breaking of direct dependencies, the gateway pattern has another advantage. It is a clear point at which to deploy a *Service Stub* [30, p. 504-507]. When testing single components, the dependency to the middleware hinders the test cases significantly as well as increases the

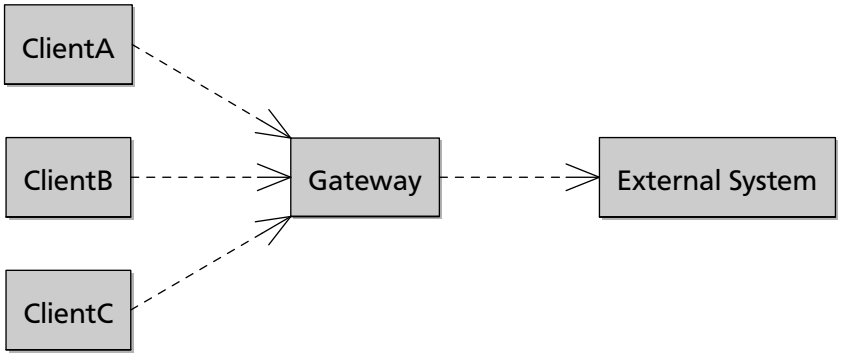


Figure 3.16: The Gateway pattern

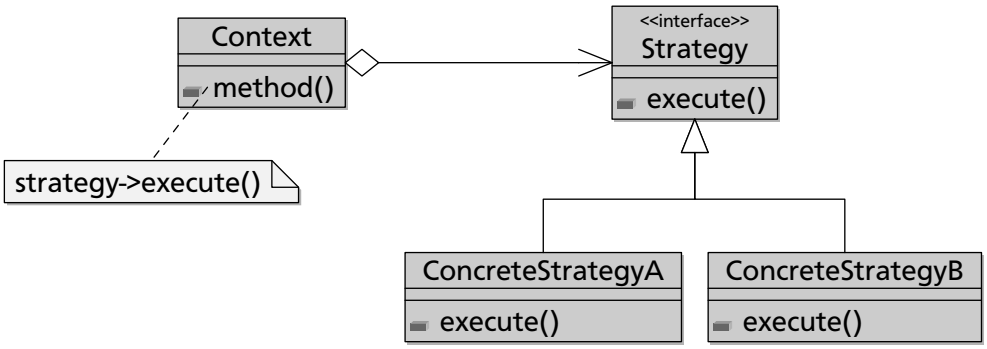


Figure 3.17: The Strategy pattern

complexity of the test. Therefore, a service stub can be used as a simplified gateway, which does not make use of the middleware at all.

The *Messaging Gateway* pattern [48, p. 468-476] is a specialization of the gateway pattern in the context of messaging solutions. It describes the possibility to encapsulate the asynchronous nature of the messaging interaction and exposes only a blocking API even if the communication is event-driven internally. For this case the gateway simply mirrors the middleware interface and therefore does not require the functionality of a messaging gateway.

Strategy Pattern

By replacing the particular gateway with a different implementation it becomes possible to alter the behavior between the component and the middleware. This concept follows the *Strategy* pattern as depicted in Figure 3.17 [39, p. 315-323]. Thus, the actual implementation of the strategy (e.g. how to publish data) becomes independent of the component itself. Still the gateway or strategy is used inside the component and is not yet customizable from the outside.

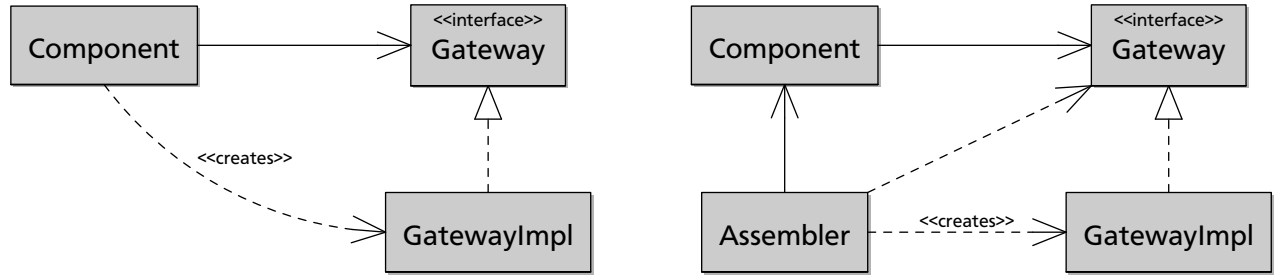
Dependency Injection

The component usually obtains access to the middleware by direct creation or usage of singletons or static functions (Figure 3.18a). To get rid of these internal dependencies, the gateway must be configurable from the outside. Therefore, the *Dependency Injection* pattern [32] is applied with the task of separating configuration from use as illustrated in Figure 3.18b.

This is also known as *Inversion of Control* (IoC) or the *Hollywood-Principle*: *"Don't call us, we'll call you."*

Three different types of injection methods are defined as:

- *Interface injection* (type 1 IoC)
- *Setter injection* (type 2 IoC)
- *Constructor injection* (type 3 IoC)



(a) The component directly creates the particular gateway

(b) The particular gateway is created from the outside and injected into the component

Figure 3.18: Dependency injection is used to decouple a component from the specific gateway

The interface injection is similar to the setter injection, though the method for injection is defined in an interface to make the injection public. The choice between the different variants is dependent on the particular usage. Generally, the type 3 injection is preferred, as it guarantees a correctly initialized instance. For the use case of loading components dynamically, it is sometimes not possible to use constructor parameters, as the framework responsible for loading the dynamic library carries out the creation of instances as well.

The implementation of most of the described concepts will be based upon the presented dependency injection method, to have the possibility of altering the strategy of the gateway between the components and the middleware.

This concept of a revised middleware interface that enables improvements to the communication efficiency, will be presented at the IROS conference 2010 [101].

4 Efficient Communication Mechanisms

This chapter deepens the concepts of the presented communication mechanisms and their realization. The improved message transfer discussed in Section 3.3.1, using references and filtering data directly at the publisher, is implemented for two different middleware, namely RoboFrame and ROS. The proposed methodologies for recording arbitrary messages locally on each robot and the team communication between multiple robots are described. Finally the realization of a bridge to interconnect both middleware is presented, which enables the exchange of messages between them.

4.1 Message Exchange by Reference

4.1.1 Interface Design of RoboFrame

The design of RoboFrame varies significantly from other middleware implementations. It adheres to the concept of a framework rather than that of a library (Section 2.2.4).

Due to the inversion of control a component does not actively call library functionality to publish information. Instead it uses a component internal storage class for each exchanged kind of message. During initialization the components provide this storage to the framework describing its set of exchanged messages (Figure 4.1).

In the following the framework passes each incoming message to the respective storage, which is responsible for notifying the associated component. Accordingly, when a component pushes a message to a storage, the framework is notified and can distribute the data to other subscribers using the communication mechanisms. The components as well as the framework only interact with this storage for receiving and sending messages (Figure 4.2).

The storage acts similar to the gateway mentioned in Section 3.3.6. But instead of relaying the calls from the components to RoboFrame directly, it maintains the *inversion of control* specific for a framework. As depicted in Figure 4.2 the components do not publish the data themselves. Instead, the flow of control is left to the framework, which fetches the information from the storages and performs the further handling of the data.

4.1.2 Consecutive Execution of Components

Exchanging information between components usually involves object marshaling, memory copy and object demarshaling, as these components are running in separate threads, processes or even on

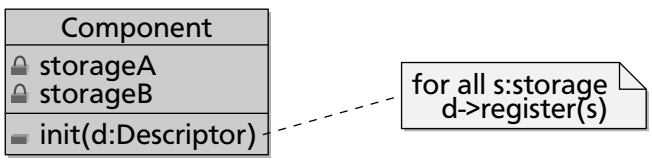


Figure 4.1: RoboFrame components provide descriptive information of the exchanged messages

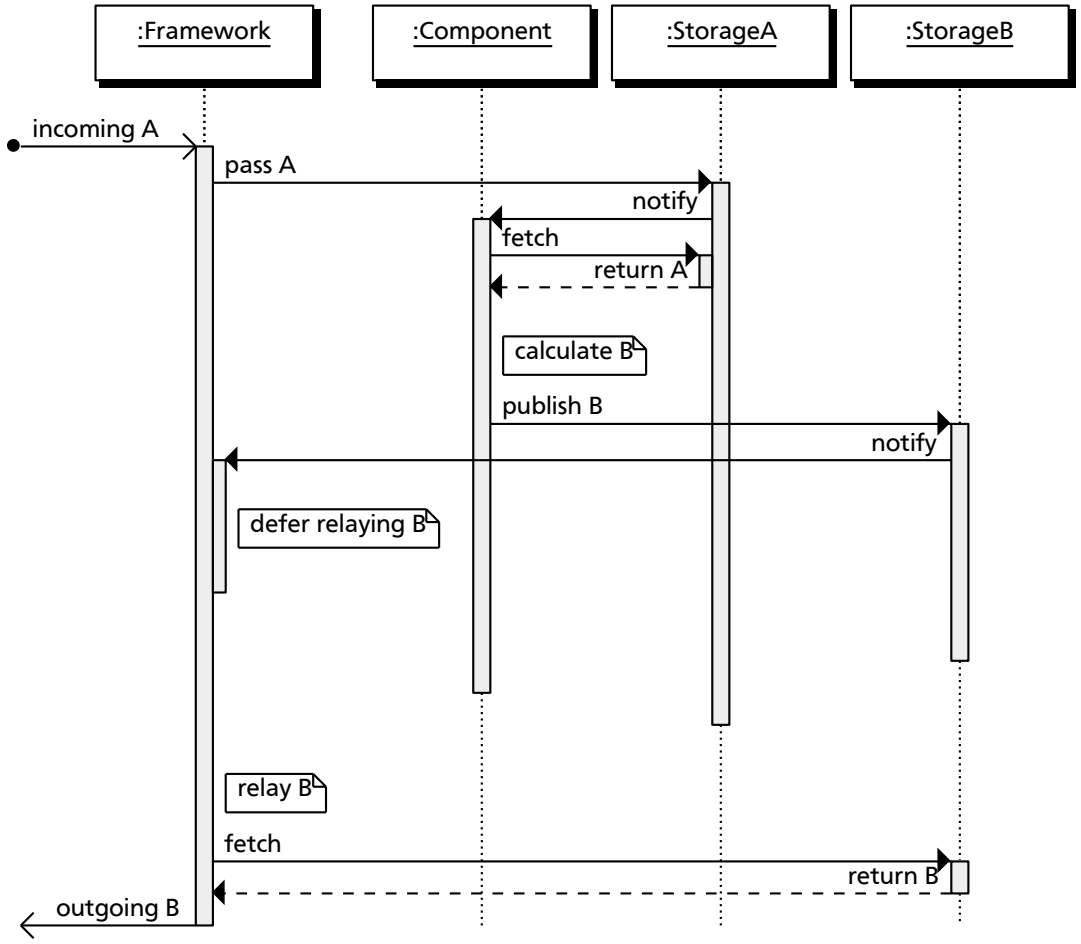


Figure 4.2: Sequence diagram of the message exchange between RoboFrame and a component. The data is passed through a storage class which acts as a gateway and maintains the inversion of control

different machines. But in cases where concurrent execution of these components is not required and they are running on the same machine, this overhead can be avoided. These assumptions usually hold for many tasks. A common application is the sensor data acquisition with the subsequent sensor data processing.

When the components are executed consecutively it becomes possible to only pass references of the messages between them. Due to the descriptive nature of the interface in RoboFrame, this goal can be achieved without compromising the decoupling between the components and still be fully transparent. Only two constraints must be followed. First, the messages must not be altered after publishing. Second, the received message must also not be modified, which can be easily ensured through an appropriate definition of the programming interface.

Therefore, the storages used in the components do not actually provide a storage, but act as a proxy for the real storage. During the initialization phase RoboFrame collects the descriptive information of all components and assigns a concrete storage to each of the messaging proxies. When multiple components are combined in a single thread all proxies for the same type of messages share the same storage. Thus, all components refer to the same instance of data when accessing the messages through the proxies (Figure 4.3).

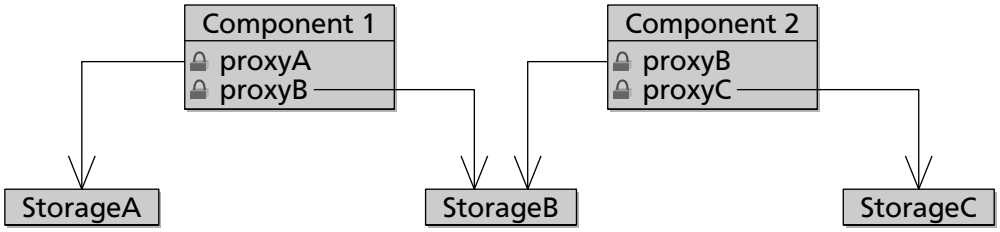


Figure 4.3: Multiple components share the same message storage through proxies

The separation between the proxies in the components and the concrete storages in the middleware constitute a special strategy according to Section 3.3.6. The impact on performance is apparent as the overhead is close to zero, and for this reason outperforms any approach utilizing any kind of copy operation. A detailed evaluation of the reduced overhead and a comparison with the performance of other approaches is covered below in Section 6.2.

Yet, the class encapsulating the interaction between RoboFrame and the components could not be exchanged from the outside due to the tight coupling of the storages with the middleware.

4.1.3 Enabling Different Strategies of Interaction

For enabling the application of different strategies, not known by the middleware, the coupling between the framework and the storages is removed. Therefore, the creation of the concrete storages, which was initially handled by the middleware, is moved to the proxies of the concrete storages. These proxies provide only an abstract interface for the framework to exchange messages, which enables to transparently use different types of storages.

During the configuration phase of a RoboFrame application, the class realizing the interaction between the middleware and the proxies can optionally be replaced with a custom implementation if necessary. This modification can be made on a per-component base.

4.1.4 Application to ROS

To demonstrate the flexibility of the concept it has been applied to another existing solution, namely ROS. It has been selected due to the following reasons: It is a message-oriented middleware using a publish/subscribe paradigm just as RoboFrame. ROS has gained increasing adoption in the recent times and is inter-operable with various other existing projects. Therefore, it will be introduced in other projects at the author's group in the near future. But in contrast to RoboFrame, ROS is designed as a library, not as a framework.

A *Node* in ROS corresponds to a component. Each node is built as a separate independent executable. Therein, during the initialization a node handle is created, which is analogous to a gateway. Any subsequent calls for advertising and subscribing to different messages is done using this handle. Afterwards the node waits for incoming data, which are handled through the registered callbacks. An example of this cycle is shown in Listing 4.1.

With the original design of ROS, hooking into the communication of an existing node is impossible.

Listing 4.1: The main function of a ROS node

```
1 int main(int argc, char** argv) {
2   ros::init(argc, argv, "my_node_name");
3   ros::NodeHandle nh;
4   ros::Publisher pub = nh.advertise(...);
5   ros::Subscriber sub = nh.subscribe(...);
6   ros::spin();
7   return 0;
8 }
```

Custom Node Interface

To achieve the aimed goal, the gateway must be made exchangeable with different strategies. Instead of letting the node create its own node handle (line 3 in Listing 4.1) it must be injected into the node from the outside. Therefore, the logic of the node except the creation of the node handle is wrapped inside a class. The concrete node handle is consequently injected as a parameter from the caller of the main method as seen in line 5 in Listing 4.2.

Listing 4.2: A ROS node wrapped in a class featuring dependency injection for the node handle

```
1  class MyNode {
2    MyNode(int argc, char* argv[]) {
3      ros::init(argc, argv, "my_node_name");
4    }
5    int main(ros::NodeHandle& nh) {
6      ros::Publisher pub = nh.advertise(...);
7      ros::Subscriber sub = nh.subscribe(...);
8      ros::spin();
9      return 0;
10   }
11 }
```

Correspondingly, the global main function is only used to instantiate the concrete node and create a custom node handle, which is then injected into the node as stated in line 3 in Listing 4.3. This main function works well with any node implementing the same interface.

Listing 4.3: Modified main function injecting a custom node handle into the ROS node

```
1 MyNode n(argc, argv);
2 CustomNodeHandle nh;
3 int rc = n.main(nh);
4 return rc;
```

Using a custom node handler, the behavior of the publish/subscribe mechanism can be arbitrarily altered. As long as the node implementation is not aware of the custom node handler, it cannot utilize any additional interface that the custom handler may provide.

ROS Nodelets

In August 2010 the ROS project has released a secondary interface beside nodes, named *Nodelets*. They are intended to provide a way to run multiple subtasks in the same process without any transport overhead for exchanged messages. However, they are only available in the C++ client library. A set of nodelets is contained in a nodelet manager, which acts as a normal node. Between these nodelets Boost shared pointers are passed by the nodelet manager. Only if a subscriber for the data outside of that node exists, the messages need to be marshaled and copied.

The purpose is similar to the developed solution in this thesis. The nodelet manager acts as a gateway and implies that the published data are not manipulated after they have been published. The received data are invariant alike. Additionally, the approach of ROS features the ability to distribute the execution of multiple nodelets on multiple threads.

But the design of a nodelet is notably different from nodes. Both kinds are handled independently and the implementation of both varies in several details. Therefore, it is not possible to use both paradigms interchangeably.

4.2 Throttle Messages at Publisher

The concept of throttling messages directly at the publisher instead of after receiving at the subscriber has been proposed in Section 3.3.3. In the following, a set of common types of throttles is presented and the details of the implementation are described.

4.2.1 Exemplary Throttling Options

In the considered scenarios different types of throttling have been applied. They depend on the particular use cases and are by no means an all-embracing set. But additional throttling configurations can easily be added by extending the presented implementations.

The following throttling configurations have been realized (Figure 4.4):

- *Every N-th message* — the use case for this throttle is to reduce the amount of messages received per time interval. It is often applied when the available bandwidth is limited. This may be the case for remote monitoring tasks as well as for recording data locally on the robot.
- *Message at most every N ms* — its use case is similar to the previous. It skips any message within N milliseconds of the previously published message. However, it is especially useful if the frequency of the publications is not known or varies over time.
- *First N messages* — is used for receiving only a fixed number of messages. After these have been sent the subscription is virtually nullified. This is often used to fetch only a single message of a particular type, for example to fetch configuration data for remote adjustment or record information only once.

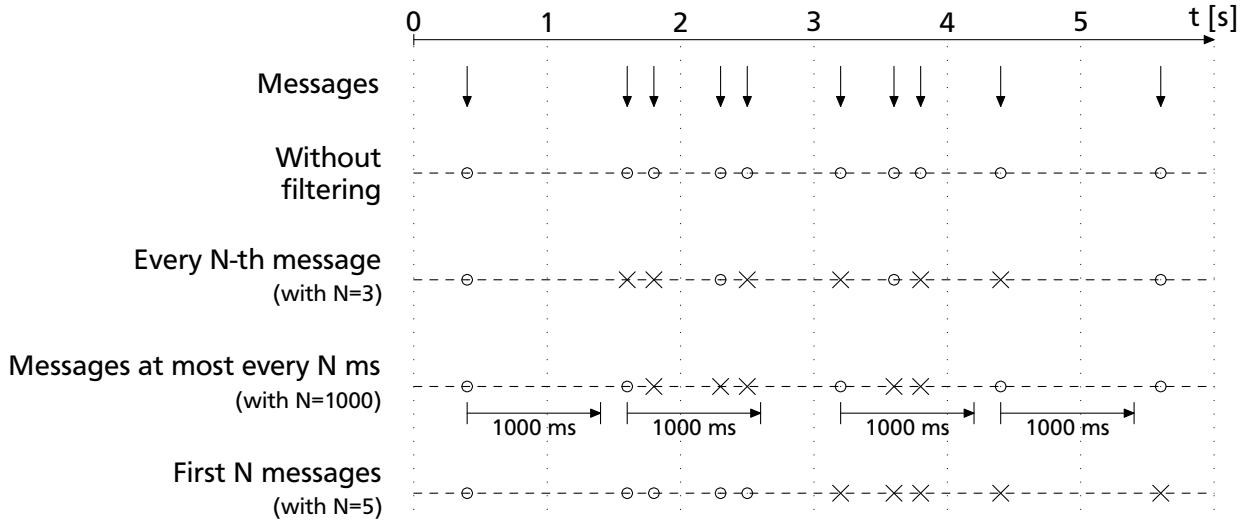


Figure 4.4: Implications of different throttling configurations
(dots indicate published messages, crosses depict skipped messages)

Passing Throttle Configurations

Instead of skipping the received messages at the subscriber, they are already *throttled at the publisher side*. Therefore, the type of desired throttling, as well as its parameters, need to be transferred to the publisher. This is done together with the arguments required for subscribing to the messages. In order to permit the specification of the throttle configuration for the components the interface of the communication gateway must be extended accordingly.

For RoboFrame, the signature of the relevant method for subscribing to messages has been expanded to support an optional configuration parameter specifying the throttling. As the framework has been developed in the author's group, the additional information has been fully integrated into the publish/subscribe mechanism.

For ROS another course of action has been selected. The interface for providing the additional throttle configuration has been extended similarly using a custom node handle as described in the previous section. But this information could not be integrated into the existing communication mechanisms easily.

Hence, the information is transported to the publisher using an anti-parallel topic as illustrated in red in Figure 4.5. This additional topic is advertised from the subscriber when a subscription with a throttling configuration is performed. On the publisher side the same additional topic is subscribed to, even if it is yet unknown if throttle parameters will be passed.

4.2.2 Throttling at Publisher

In RoboFrame the publishing component can easily check whether a specific message type is requested by any subscriber. To permit this, internally a list of all active subscriptions is maintained. In order to honor the additional throttling this optional information is attached to the particular subscription entry. With this information available at the publisher side the current configuration can be evaluated to check if a message is requested right now.

The event, when a message should be published, regardless of whether it is skipped or actually passed to any subscriber, is called *virtually published* in the following. For each virtually published

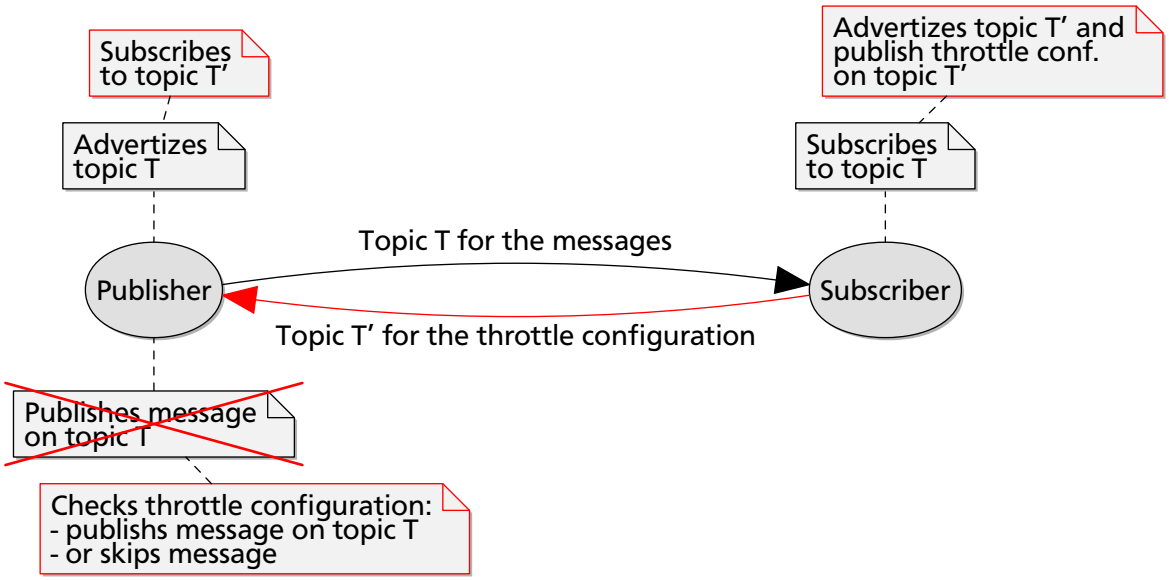


Figure 4.5: For ROS the throttle configuration for topic T is passed through an anti-parallel topic T', which is automatically advertised on subscription and enables the publisher side to directly apply the throttling

message the throttle configuration for each subscriber is considered, to decide to which receivers the message has to be forwarded to. Additionally, these throttle configurations need to be updated for every virtually published message. For example, for a throttling of every n-th message an internal counter must keep track of the virtually published messages, in order to pass every n-th message to the subscriber. The required information depends on the particular type of the throttle configuration.

Implementation on Top of ROS

In ROS the implementation of the throttling at the publisher side was much more complicated. The actual publishing to individual subscribers is handled by the *Publication* class, which is responsible for passing the messages to the particular *SubscriberLinks*. In order to permit throttling, the behavior of the publication instance must be manipulated. As a custom NodeHandle can be injected into the publishing node as described above, it has been used to manipulate the complete hierarchy down to the publication class as depicted in Figure 4.6.

Several issues of the ROS implementation (marked with red annotations) made it necessary to alter the ROS classes. The modifications are limited to changing the visibility of member variables and methods from private to protected, defining several methods as virtual in order to permit overriding them in subclasses and make internally created instances exchangeable by subclasses as marked with the green annotations.

The custom hierarchy is necessary to provide custom instances of each class in the chain. The CustomNodeHandle provides a CustomPublisher instance, which itself needs to provide a CustomTopicManager, which again provides a CustomPublication instance as illustrated in Figure 4.7.

However, except for the CustomPublication class, the other custom implementations are only thin wrappers on top of the existing classes. The publish-method of the custom publication class can be used to implement an arbitrary behavior for passing the messages to the subscriber links.

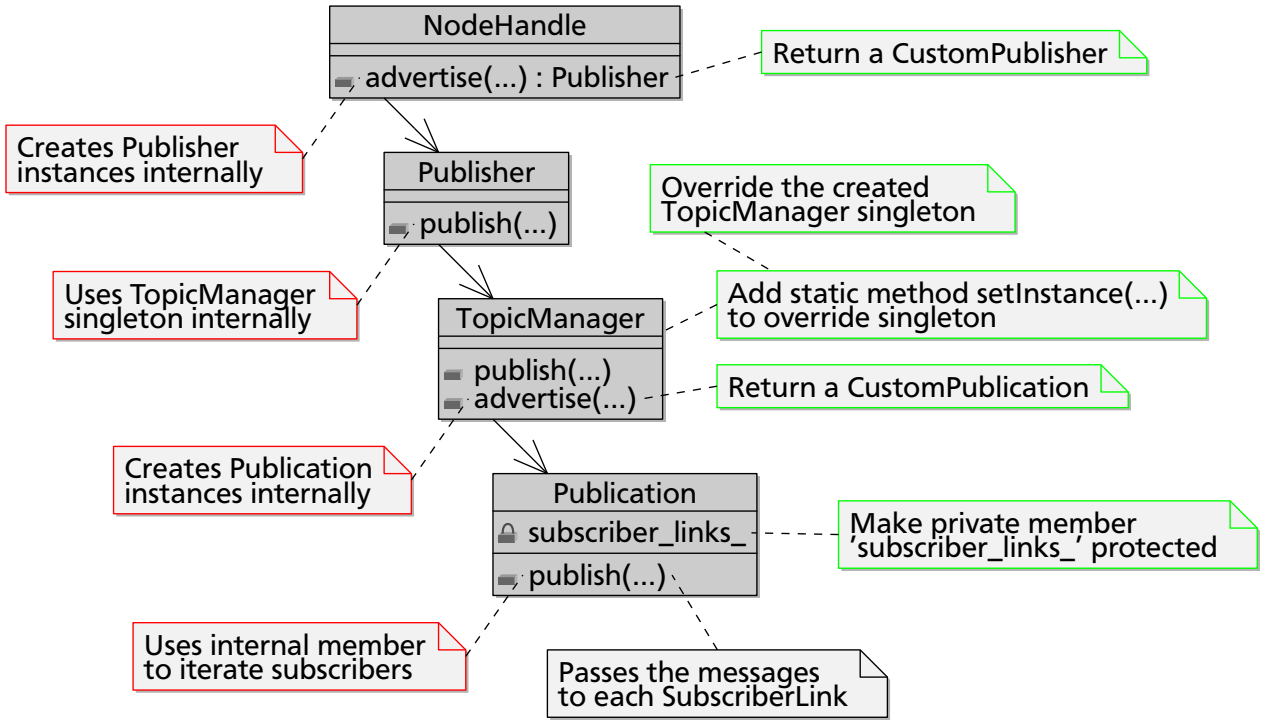


Figure 4.6: The classes involved in the procedure of publishing messages in ROS. Inheriting from these classes is problematic as they use private variables and methods as well as singletons and create instances internally

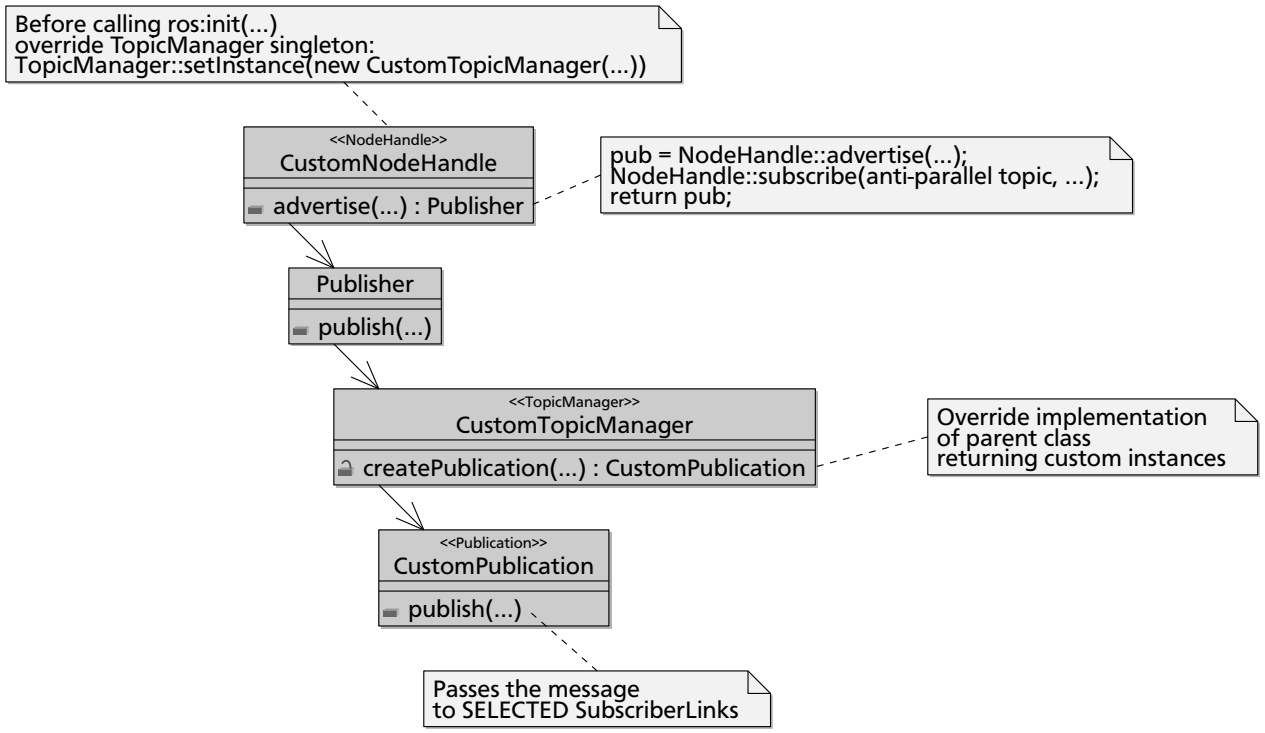


Figure 4.7: The custom classes injected into ROS are based on a custom NodeHandle and a custom TopicManager singleton. The custom Publication instances alter the behavior of the message dispatching to the subscribers according to the received throttle configuration

In this case it has been used to realize the throttling, which passes the messages only to a subset of the subscribers according to the current throttle configuration.

Example Illustrating Increase in Efficiency

By making the information about the subscriptions and throttle configurations available to the publishing component, any computation for later skipped messages can be avoided.

A prime example for this functionality is the following use case: A remote graphical user interface requests the images of the camera. But in order to keep the required bandwidth low, only JPG-compressed images are requested with a frequency of once every second.

The advantage of less used bandwidth due to the throttling on the publisher side is obvious. But due to the provided meta information to the publishing component, which encodes the raw images into JPG format, the overhead of encoding can also be avoided for virtually published and skipped images.

4.2.3 Differentiating Multiple Subscribers

When multiple components are subscribed to the same publisher the throttle handling becomes even more difficult. The decision if any subscriber is currently requesting a message according to its respective throttle configuration is straightforward. The result of each single subscriber must be combined with an OR-operation.

But the result of each single subscriber must be remembered for deciding to which subscribers the messages have to be sent. For the ROS implementation it is unproblematic, as the communication layer keeps a discrete connection to each subscriber.

As RoboFrame follows the pattern of a central message broker [48, p. 322-326] The differentiation, which subscribers receive the message and which do not, becomes a challenging task (Figure 4.8). For the central message broker it is impossible to decide to which subscribers a message should be forwarded to. The throttle configuration cannot be evaluated by the message broker, as this implies a second throttling operation in addition to the already performed operation at the publisher.

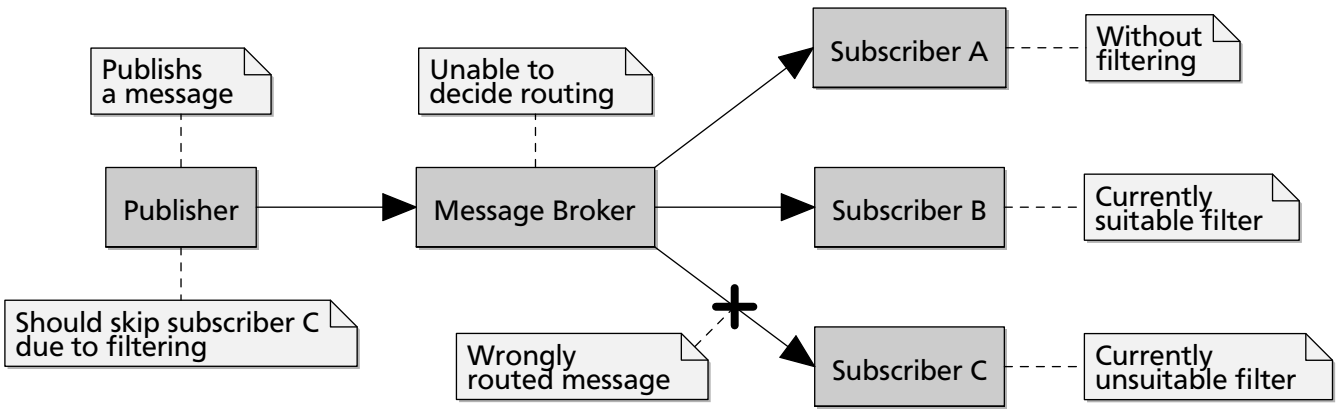


Figure 4.8: Messages cannot be routed correctly by the message broker when throttling is applied

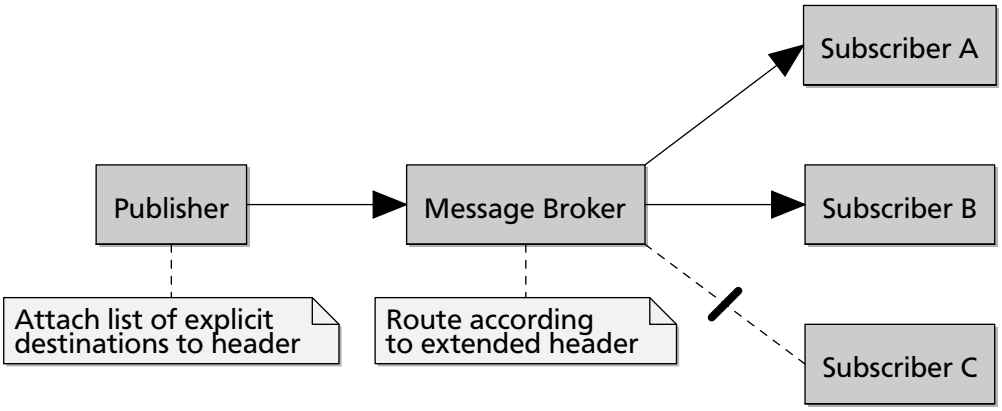


Figure 4.9: The message broker uses the extended header for routing the message to only a subset of the subscribers

Instead of changing the communication methods to a peer-to-peer communication, and sending the message to each subscriber individually, the message is still passed only once to the central message broker, to avoid a multiplication of the exchanged data. For enabling the message broker to decide about the forwarding of the message, it is marked with a list of explicit destinations at publication time. Therefore, the header of the communicated packet containing the message is extended with a variable list of destinations, which can be utilized to perform a custom routing in the message broker as depicted in Figure 4.9.

This solution does not scale well for a considerable number of subscribers as the list of destinations added to the header of the message would increase linearly. But for the aforementioned scenarios this is not an issue as the number of subscribers is manageable.

4.3 Recording Intrinsic Data on Robot

In order to enable offline analysis, the intrinsic information of the robot control software must be recorded. Later on this information can be played back for offline introspection and analysis or fed to another instance of the control software for repeatable testing with identical input values.

The complexity of the considered applications induces a tremendous amount of data exchanged between the different functional components. Detailed measurements of the quantity of messages and their sizes are given in Section 6.2. Because of the strictly limited bandwidth of the wireless network usually applied for the considered applications this information has to be recorded locally on each robot.

4.3.1 Intrinsic Data

As the application software is based on a message-oriented middleware, all intrinsic data are passed as messages between functional components. This makes it easy to subscribe to every relevant message bus and store the messages, e.g., in a logfile. But the amount of exchanged data is quite significant, in particular for raw image data. For example, a single uncompressed RGB image with VGA resolution requires approximately one Megabyte. At a frequency of 30 fps the data rate is more than one and a half Gigabyte per minute.

The used hardware platforms have two limitations relevant to the local data recording: First, the overall space for storing information is limited to a few Gigabytes; second, the throughput to

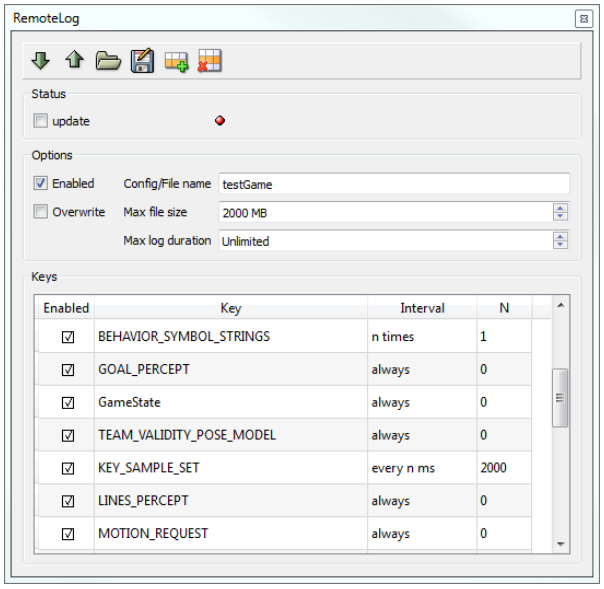


Figure 4.10: The dialog to remotely configure and trigger logging functionality directly on the robot

write data to the storage is quite limited, so that the amount of data stored per second must be reduced.

Depending on the debugging focus, various different subsets of information are relevant. The type of messages to record must be configurable as well as their frequency. Therefore, the previously described feature of efficient throttling is utilized to avoid additional overhead created by receiving messages which are actually not recorded.

The functionality is implemented as a component of RoboFrame. The subset of messages to be recorded can be configured at deploy time with the recording beginning automatically at the start-up of the application. Alternatively, the subset can be configured and the recording be started at runtime using a remotely connected graphical user interface (Figure 4.10). While the first method is applied for field operation in the considered scenarios, the second is utilized when conducting testing and debugging tasks.

Furthermore, some usability features have been integrated, i.e., stopping the recording when the storage device is nearly full and limiting the amount of data to a specific maximum log size.

In Section 6.3 measurements are presented which focus on the performance impact on the overall application when recording different amounts of intrinsic data.

4.3.2 Synchronization

For analyzing the distributed recorded messages from multiple robots, the data must be synchronized, as it cannot be guaranteed that the different hosts share a global time. Instead of performing the synchronization manually for every single logfile, a method to determine the time offsets between the teammates has been implemented.

Therefore, each robot sends a broadcast of its current time and responds to other broadcasts of teammates with a unicast to the sender. This broadcasting is performed continuously every N seconds in order to enable dynamic addition of new robots to the team and compensate for unreliable network conditions. The procedure is derived from the simple network time protocol (SNTP) [75]; the temporal data flow of the exchanged messages is shown in Figure 4.11. But

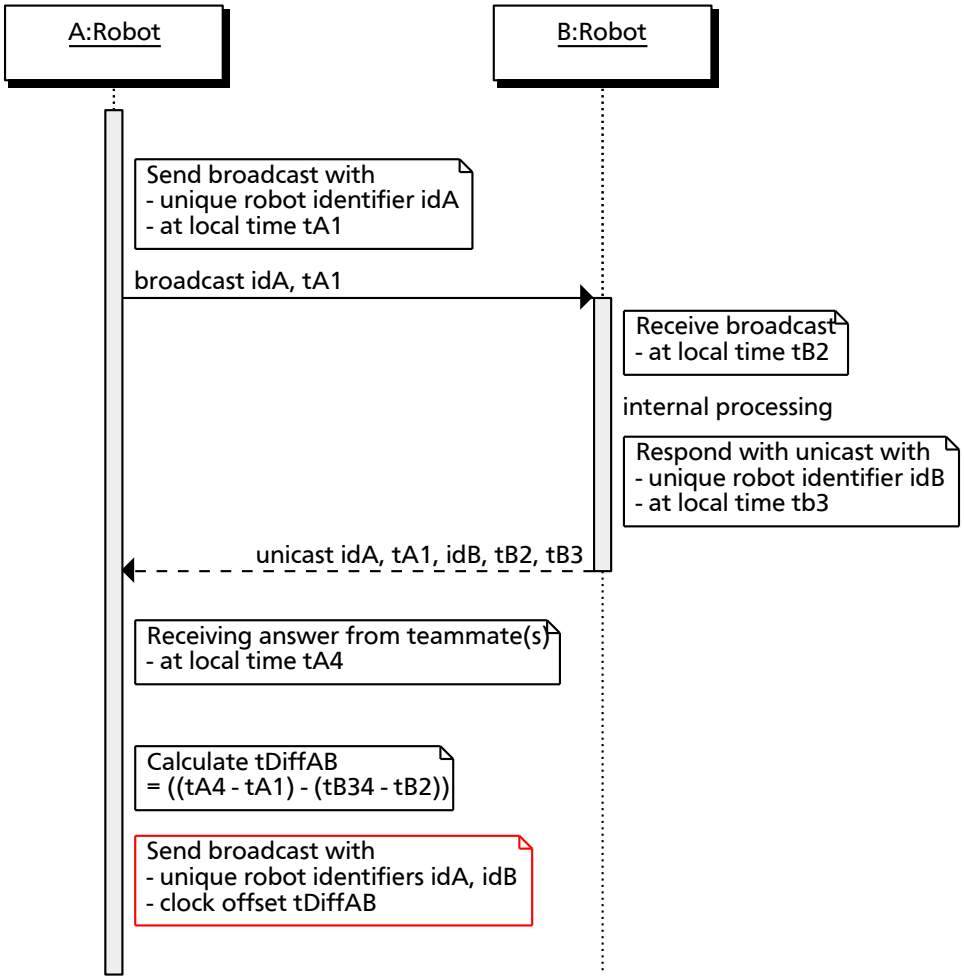


Figure 4.11: Sequence diagram to determine the clock offset between two hosts. The result is broadcasted to inform other teammates about the calculated offset between these two robots

instead of adjusting the system time of one host, the calculated offset between two hosts is recorded to the logfile for offline interpretation.

This information is sufficient for automatically synchronizing logfiles of multiple robots, as long as each robot is aware of the clock offsets compared to other teammates. But in some situations this assumption does not hold. Figure 4.12 illustrates a common situation in the soccer scenario. The clock offset between the robots A and B is known to both of them and for the couple B and C. However, since robot A and C have never been running simultaneously, the offset between their clocks is unknown. Only if the recorded data of all three robots is considered, a relation between robot A and C can be established via robot B.

In order to permit the automated synchronization, even if only the recorded information from the robots A and C are considered, the known time offset between any two robots is additionally broadcasted to all other teammates. This assures that each robot is aware of all time offsets as illustrated with the red boxes in Figures 4.11 and 4.12.

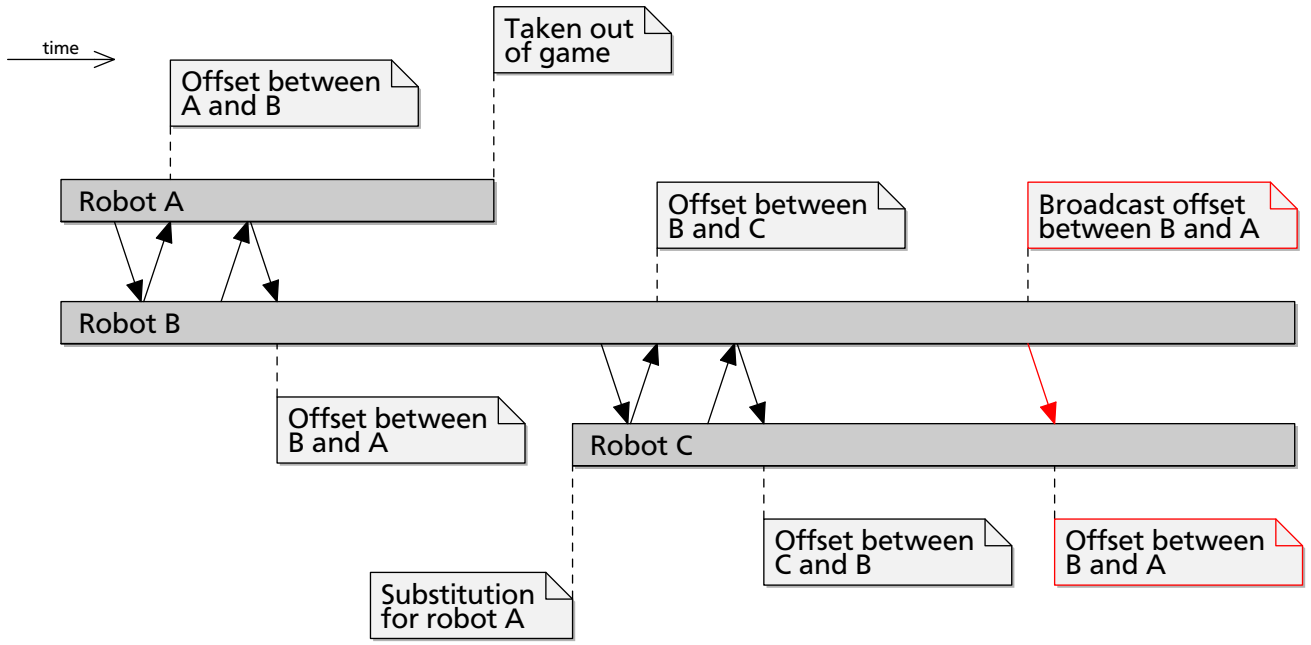


Figure 4.12: Clock offset calculation and propagation for multiple robots

4.4 Communication Between Multiple Robots

At the first look UDP broadcasts seem to be a reasonable choice to exchange information between multiple robots efficiently. But as stated in Section 3.3.5 multiple, different scenarios have to be addressed. For two of these the usage of broadcasts is inapplicable. First, if multiple instances of the robot control software are executed on the same host, e.g., when testing the team behavior of multiple robots in simulation, each instance must use a different port for incoming data. Second, when multiple developers are working independently of each other but on the same subnet, the team communication might unintentionally affect the other operations.

The other alternative, with the same property of sending data only once, is the usage of multicasts. For communicating over a wired network based on the Ethernet standard, the choice would be reasonable. But due to their mobility the robots use wireless network communication according to the IEEE 802.11 standard [50].

As mentioned in [18] the rate adjustment on the wireless network is limited to unicast traffic only. As a consequence, multicast as well as broadcast traffic is always transmitted at the lowest possible rate. The restricted bandwidth is not a primary issue. But utilizing the lowest possible rate results in longer periods of time for transmitting the same amount of data, which increases the probability of collisions and thereby loss of datagrams.

Thus, the exchange of information between multiple robots is realized with UDP unicasts, even though this implies sending datagrams to each single teammate individually. To avoid additional traffic during operation, the list of potential teammates and their IP address and port can be configured at deploy-time.

Specific Issues on Windows

During the implementation and testing of the unicast based information exchange, two severe problems have been encountered.

First, when sending larger UDP datagrams, which need to be fragmented due to the maximum transmission unit, from a Windows XP host without Service Pack 3, the IP checksum value may be calculated incorrectly. Therefore, the receiving host may identify the UDP packet as corrupted and discard it. This issue has been addressed in a hotfix available at [74].

Second, according to the standard, packets sent to the global broadcast IP address 255.255.255.255 should be sent to the Ethernet global broadcast address (ff:ff:ff:ff:ff:ff) on all available interfaces. Neither Windows XP nor Windows 7 works correctly; both even exhibit different behavior about this [94]. Windows XP sends the packets via the first available network adapter only. For other than the first adapter global broadcasts are therefore impossible. In Windows 7 the particular adapter used for global broadcast can at least be configured using the routing table.

Since almost every modern computer has two network interfaces, e.g., for cable and wireless, the issue of Windows limiting the broadcasts to a single network adapter is severe.

Exchange of Heterogeneous Data in Teams of Robots

The unicasts present a basic message based communication. But it requires an agreement on exactly what data are exchanged between all participating agents.

For heterogeneous robots this assumption is not valid, since their world models might be heterogeneous too and do not necessarily share the same definition.

Instead of unifying the models to the lowest common denominator, another approach has been developed [93]. The information of the world model is split into elementary units, which are uniquely named and mostly correlate with the primitive data types. At the beginning of a cooperative task each robot communicates the definition of its model containing the unique names of all units along with their particular type. Every robot can then compute the globally shared model definition based on the received model specifications and its own definition by overlapping the different sets of units (Figure 4.13).

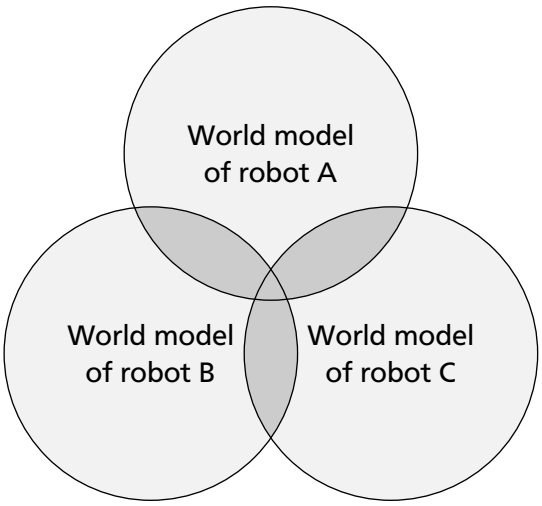


Figure 4.13: The calculation of the global world model is performed on each robot and consist of the overlapping of all model definitions in the team

After the global world model has been calculated each robot sends all units of its own model, which are shared with at least one other agent. The receiver of the team communication must be able to skip units contained in the exchanged messages, which are unknown in its own model.

This approach provides a very flexible system, where heterogeneous robots with different world models can easily exchange common information without the demand for exact equality of the models.

Example scenarios of heterogeneous teams of cooperating autonomous robots are mentioned in detail in [59]. For the scenarios considered in this thesis the application for exchanging information between teammates on different levels of the control software as well as the utilization for online monitoring is described in Section 6.4.

4.5 Bridging Messages Between RoboFrame and ROS

The recent developments of ROS are very promising and may lead to a wide adoption. Especially, due to the ability to use different programming languages, it is also contemplated for the author's group. On the one hand, it is used in new projects like the BioBiped [1]. But on the other hand, it is also considered to be jointly used with RoboFrame for the described RoboCup scenarios. Some components are transfered to ROS in order to exploit the advantages of other programming languages. For example, a Java implementation of the XABSL engine reduces the efforts required to link the behavior with various input data due to the usage of reflection as described in [85].

However, the extend and complexity of the already developed software cannot quickly be converted to a new middleware. Instead, an integration between both systems is reasonable to enable interoperability. Therefore, it is necessary to provide a mechanism to exchange information between these two communication systems. This corresponds to the design pattern of a *Messaging Bridge* [48, p. 133-136].

Commonly the main issue is that both messaging systems use different methodologies for storing their messages. For this reason it is typically not possible to connect two complete messaging systems. Instead individual channels can be linked.

The messaging bridge pattern has been applied, which consists of a set of *Channel Adapters* [48, p. 127-132]. Thereby, the non-messaging client of the channel adapters is actually the other messaging system. Therefore, two unidirectional channel adapters together form a bidirectional bridge between a pair of corresponding channels in both messaging systems (Figure 4.14).

[1] http://www.biobiped.de

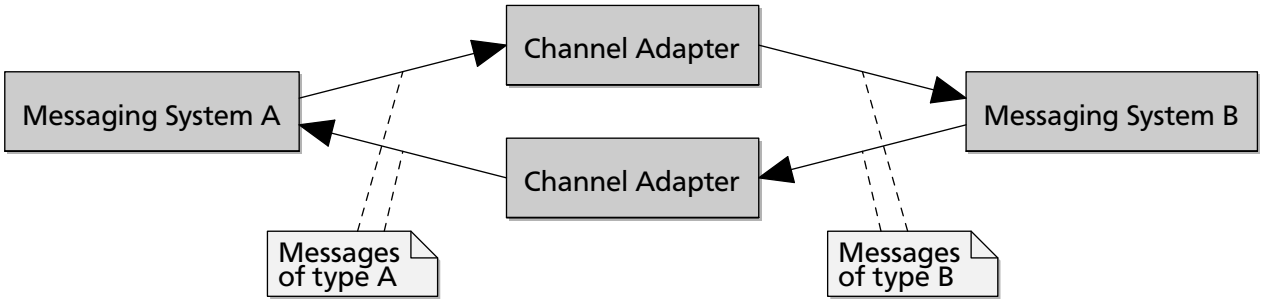


Figure 4.14: A messaging bridge consists of a pair of channel adapters connecting corresponding channels of two messaging systems

The link-up with the channel adapters could be realized from either side. First, by developing a ROS node which connects to the message broker of a RoboFrame application. Second, by implementing a RoboFrame component which utilizes the ROS library to exchange information. Due to the distinct nature of both solutions it is more difficult to integrate the message format of the framework RoboFrame into ROS than to use the ROS library in a RoboFrame component. Thus, the second variant is the more promising approach and therefore has been followed.

RoboFrame Component for Interfacing with ROS

For both directions a channel adapter is implemented as a RoboFrame component. While running as a part of the RoboFrame application, it also acts as a ROS node to exchange information. As the channel adapters need to be aware of the concrete message types, they are realized as templates (line 2 in Listings 4.5 and 4.6). Besides the types of the messages the particular identifiers of both channels, which should be linked, must be configured using constructor parameters (see line 8 in both listings).

The initialization of the ROS node and either the advertisement or the subscription are carried out in the start-up phase (*init*-methods). The tear-down is handled accordingly (*cleanUp*-methods). Handling the transformation between both message types is more challenging.

Neither of the message classes need to be aware of the other in order to avoid introducing any dependency between them. However, the messaging bridge needs to be generally applicable. Hence, it adopts the template method pattern [39, p. 325-330] and implements the invariant behavior of the channel adapters. But contrary to the pattern, the hook operation is not provided through subclassing. Instead, an external transformation function is utilized (line 29 in Listing 4.5 and line 31 in Listing 4.6). Providing such a transform implementation requires no integration with either of the messaging systems (Listing 4.4).

Listing 4.4: Transformation of a RoboFrame streamable into a ROS message

```
1 // the function depends on the types of both messsages
2 void transformRoboframe2Ros(
3   const RoboframeStreamable& src,
4   RosMessage& dst
5 ) {
6   // performs any kind of mapping from source to destination
7   // e.g.
8   dst.data = src.data;
9 }
```

Listing 4.5: Channel adapter to accept RoboFrame messages and publish them using ROS

```
1  // the template depends on the types of both messsages
2  template<typename RoboframeStreamable, typename RosMessage>
3  class Roboframe2RosChannelAdapter
4    : public RoboframeComponent {
5
6    // the channel identifier in RoboFrame is called key
7    // the channel identifier in ROS is called topic
8    Roboframe2RosChannelAdapter(Key key, string topic) {}
9
10   void init() {
11     // subscribes to the RoboFrame channel
12     // identified by the key
13     request(key);
14
15     // initializes the ROS node
16     ros::init(...);
17     nh = new ros::NodeHandle();
18
19     // advertises the RosMessage at the channel
20     // identified by the topic name
21     pub = nh->advertise<RosMessage>(topic, ...);
22   }
23
24   void relay(const RoboframeStreamable& data) {
25     // transform RoboFrame streamable into ROS message
26     // using an external transformation implementation
27     // avoiding any dependency between both message classes
28     RosMessage msg;
29     transformRoboframe2Ros(data, msg);
30
31     // publish it via ROS
32     pub->publish(msg);
33   }
34
35   void cleanUp() {
36     pub->shutdown();
37     delete nh;
38
39     revoke(key);
40   }
41
42 };
```

Listing 4.6: Channel adapter to subscribe for ROS messages and relay them to RoboFrame

```
1  // the template depends on the types of both messsages
2  template<typename RosMessage, typename RoboframeStreamable>
3  class Ros2RoboframeChannelAdapter
4    : public RoboframeComponent {
5
6    // the channel identifier in ROS is called topic
7    // the channel identifier in RoboFrame is called key
8    Ros2RoboframeChannelAdapter(string topic, Key key) {}
9
10   void init() {
11     // initializes the ROS node
12     ros::init(...);
13     nh = new ros::NodeHandle();
14
15     // subscribes to the ROS channel
16     // identified by the topic name
17     // providing a callback for incoming RosMessages
18     sub = nh->subscribe(topic, ..., &callback, this);
19   }
20
21   void run() {
22     // enters simple event loop processing callbacks
23     ros::spin();
24   }
25
26   void callback(const RosMessage& msg) {
27     // transform ROS message into RoboFrame streamable
28     // using an external transformation implementation
29     // avoiding any dependency between both message classes
30     RoboframeStreamable data;
31     transformRos2Roboframe(msg, data);
32
33     // publish it via RoboFrame
34     queueOut->push(key, data);
35   }
36
37   void cleanUp() {
38     sub->shutdown();
39     delete nh;
40   }
41
42 };
```

5 Efficient GUI Tools

In this chapter the implementation of the integrated graphical user interface is presented starting with a short discussion of interface concepts and available GUI toolkits. Afterwards, the various usability aspects are addressed with a platform independent solution, which is the foundation of the further presented specific user interfaces of RoboFrame and ROS. At the end of this chapter, extended analysis features developed in this thesis are described with a special focus on offline analysis capabilities for teams of autonomous robots.

5.1 Integrated GUI

The advantages of an integrated GUI compared to a set of separate tools have already been stated in Section 3.3.2.

The infrastructure of a GUI for robotics applications must provide the frame for custom tools and should not be coupled with a particular middleware. It includes only the following two features: *extensibility* with custom elements, which could be any graphical element or even non-graphical items like keyboard shortcuts, and *comprehensive usability* features, to make the usage of the tools as efficient as possible.

5.1.1 Different Interface Concepts

Depending on the particular task the graphical user interface needs to provide a distinct set of visualizations and controls. These can be organized in various different ways [65]:

- *Single document interface* (SDI) — each individual window is handled by the window manager of the operating system and contains its own menu bar and toolbars. For example, a typical SDI application is the Internet Explorer 6.
- *Multiple document interface* (MDI) — multiple windows reside under a single parent window. A single menu bar and the toolbars are shared between all child windows. But with an increasing number of free-floating windows the user becomes confused due to the jumble of interfaces. An example of a MDI is Visual Studio 6 as well as the initially developed GUI of RoboFrame.
- *Tabbed document interface* (TDI) — is a specialization of the MDI model. The windows also reside under a single parent window but are not free-floating as in MDI. As each window is always maximized inside the parent window, only one can be visible at a time. The others are accessible using tabs, whereby comparing information from different windows becomes more difficult. This concept is popular for modern web browsers as well as preference panes.
- *Integrated Development Environment-style interface* (IDE) — is a functional superset of MDI applications. While still residing under a single parent window, which is called the workbench, the windows can be more flexibly arranged. They can either be viewed side-by-side or tabbed for specific sub-panes. Additional enhancements like docking and collapsing windows are available. Since IDEs are the name giver, common applications are Eclipse and NetBeans.

The complexity of an integrated development environment and a GUI for robot control software is quite similar. The users working with either of these tools are equally skilled, since the same group of people is using such software. Therefore, an IDE-style interface is chosen for the integrated GUI targeting developers of complex robot control software for autonomous mobile robots.

5.1.2 Widget Toolkits

For the development of a GUI, various different tools are available. These provide a rich set of widgets and are therefore called widget toolkits. Whereas separate tools can utilize different libraries for building the graphical user interface, an integrated GUI is based on a single technology. To decide on a particular solution, which fits several different use cases well, various aspects must be considered.

Due to the heterogeneity of the operating systems used among the group of developers, all three major platforms Windows, Linux and Mac OS X need to be supported. Additionally, the robotics domain requires visualizing complex information, which includes three dimensional views of the robot and the environment, demanding OpenGL support or similar methods.

Other desired features for increased usability like global actions and shortcuts need to be provided by the used toolkit. Generally, the project needs to be actively supported and used by a large community in order to form a stable future-proof solution.

The amount of available widget toolkits is extensive[1], but due to the already stated requirements the list of candidates is reduced quickly. For example, platform specific solutions like the Microsoft Foundation Class, the Windows Presentation Foundation as well as Apple's Cocoa are not considered in the following.

The following list mentions the most viable candidates:

- *FLTK* — the fast, light toolkit uses a lightweight design and restricts itself to GUI functionality only.
- *GTK+* — the GIMP toolkit is one of the most popular toolkits for the X Windows System.
- *Qt* — is a cross-platform framework for application development. It is widely used for the development of GUIs but also provides non-GUI features including cross-platform abstractions for file handling, network support and thread management. The rich set of features for GUI includes a model/view architecture as well as an ECMAScript compatible scripting engine.
- *Tk* — is a platform-independent GUI framework for the scripting language Tcl. Thus, as Tcl, Tk is interpreted.
- *wxWidgets* — formerly wxWindows, provides a thin abstraction to a platform's native widgets. The toolkit is not restricted to GUI development, but provides additional features like inter-process communication, socket networking functionality and more.

[1] http://en.wikipedia.org/wiki/List_of_widget_toolkits

Due to the comprehensive set of features as well as the wide adoption and the consistent development, the GUI toolkit Qt has been selected. Especially the included QtScript module, providing a scripting engine based on the signals and slots concept of Qt, is exploited to provide an additional command line interface beside the graphical counterpart with minimum efforts. Furthermore, Qt supports several of the previously mentioned concepts to improve the usability out-of-the-box.

5.1.3 Developed Infrastructure

The infrastructure of the graphical user interface is *decoupled* of any particular *middleware*. The graphical application consists of a single main window, which acts as a container for all graphical widgets. It features the common concepts of menu bars, toolbars and dockable windows as provided by Qt.

The various features to improve the usability of the GUI as mentioned below in Section 5.1.4 are implemented on top of that infrastructure and are therefore available to any integrated component independently of the particular middleware. Each provides a different set of controls to the user interface, e.g., the managing of perspectives is implemented using a sub-menu in the menu bar.

For each supported middleware a *manager* handles the interaction between the GUI infrastructure and the middleware specific views and controls. These managers are responsible for providing the various different views and toolbars, which utilize the communication mechanisms of the particular middleware.

The relationship between the infrastructure and the managers is illustrated in Figure 5.1. The managers are responsible for providing middleware specific features and handling the instantiation of views and toolbars. They are realized as separate libraries, which are dynamically loaded by the

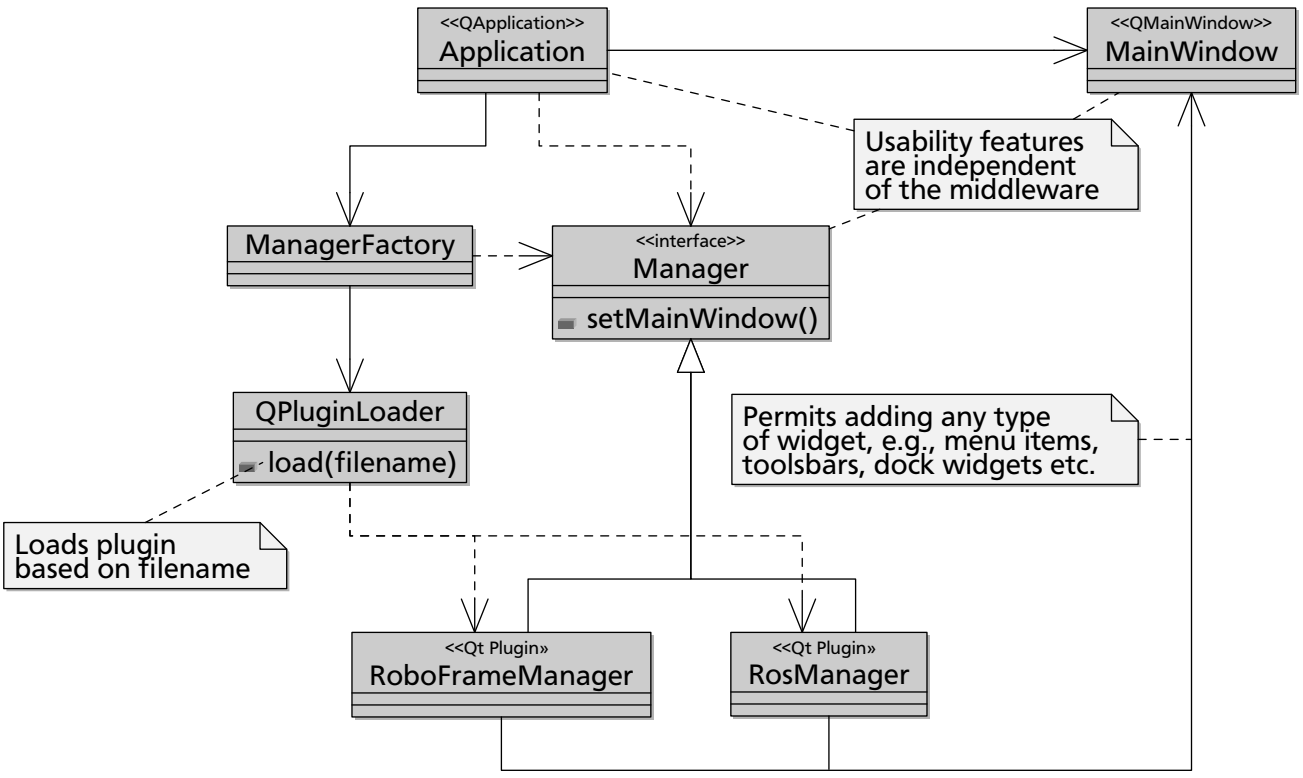


Figure 5.1: The GUI infrastructure with dynamically loadable managers for each middleware which can contribute any type of widgets to the user interface

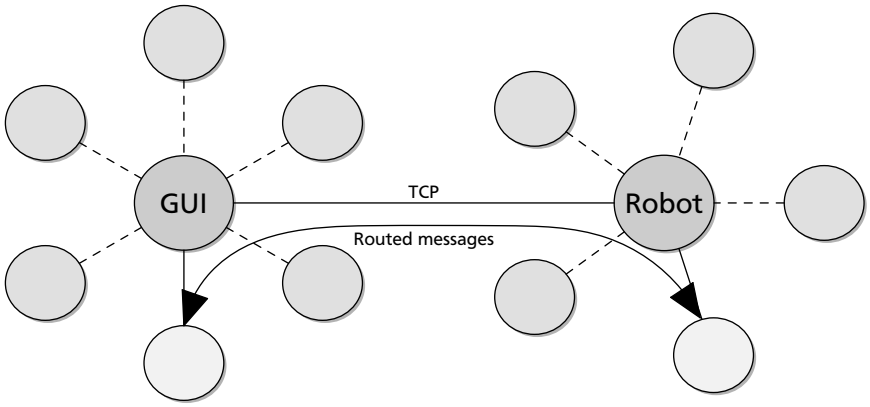


Figure 5.2: The messages of independent instances of RoboFrame are transparently exchanged through an explicitly established TCP connection between the message brokers

infrastructure. The integration of graphical components for other middleware is easily achievable, due to the strict separation between the self-contained infrastructure and the middleware specific managers.

RoboFrame

For RoboFrame the manager initially creates the *central message broker*. The available views are configured at compile time. These can be instantiated using a menu listing all registered views.

In order to exchange information with a single or multiple robots, RoboFrame provides a view which establishes a TCP connection with a remote message broker. Messages are then passed between both instances transparently as illustrated in Figure 5.2. Thus, it is not necessary to manually configure each view because the connections are handled globally. The scenarios dealing with multiple robots are described later in Section 5.3.2.

ROS

The ROS manager can either utilize an existing master or automatically create a separate instance. It starts a nodelet manager which connects to this master. In contrast to RoboFrame, the list of available views is not set at compile time. Instead, every instantiable view is implemented as a nodelet and can be dynamically loaded at runtime. The available list of views is dynamically built based on a query of the available shared libraries containing these nodelets. These views can equally be instantiated using a menu.

For exchanging messages between two masters, ROS currently provides the *foreign_relay package*. The C++ implementation supports unidirectional communication only, but allows for renaming the topics connected between both masters. A bidirectional communication can be established using two instances of this package. Since exchanging all topics automatically would include sharing high bandwidth many-to-many topics, *each message bus* has to be connected *explicitly*.

5.1.4 Features for Improved Usability

While an IDE-style interface consolidates the multiple windows compared to separate tools, it does not improve the usability in the first place. But several usability features can be implemented for an integrated GUI, which are impossible for separated applications.

Persistence for Window Arrangements

Similar to separated tools, the different views in an IDE must be opened up and arranged. For each task the compilation of tools is different, as any use case requires the visualization of particular aspects. Quickly the amount of work required to configure the user interface to the custom needs becomes annoying when used frequently.

In order to relieve the user from configuring the workbench after every start, the current window arrangement is saved and restored for the next session according to the *Autosave* pattern [102]. Such functionality is provided transparently for all GUI elements.

But the persistence needs not be restricted to the state of the windows. It also enables each view, toolbar etc. to store arbitrary intrinsic information. This may involve current selections, states, configurations and so forth, in order to release the user from restoring the exact configuration of each GUI element manually.

Switching between Perspectives

Especially when the developer is often switching between multiple tasks, the very inconvenient operation overhead of rearranging the workbench recurs. Hence, multiple different task-oriented arrangements of the workbench are stored. This concept has been established by the Eclipse project and is called *perspectives* [96]. After the initial composition, the user is able to switch between different perspectives without any need of manual reconfiguration.

Global Shortcuts

For frequently used actions the point-and-click concept of graphical user interfaces is not well-suited. Even when involving a mouse move and a single click only, other kinds of interactions are more efficient to use. As a consequence common operations need to be directly accessible using keyboard shortcuts. This feature is widely used for many kinds of software and aims for the more advanced users, which matches the targeted user group of the described scenarios. For separate tools global keyboard shortcuts are impossible since the interaction is limited to a single application currently holding the focus.

Scripting Interface

Another kind of non-graphical interface is the command line interface. A text-only interface is useful, when commands or options can be entered more rapidly than with a pure GUI. Especially, the target group of software developers is used to command line interfaces like shells. They are familiar with this concept and demand for advanced scripting capabilities in order to automate

recurring instruction sequences. Supporting such a supplementary interface increases the efficiency even further.

The developed infrastructure provides a distinct view with a scripting interface. It permits opening and closing views as well as working with the perspectives. Each particular view can contribute specific scripting functionality in order to enable custom commands for efficient interaction. Optimally every action is also accessible from the command line interface.

Evaluation of Improved Usability

All of the described features address the usability of the user interface. According to the Keystroke-Level Model it is fairly obvious that the amount of user actions is considerably reduced by the presented functionality. Most enhancements permit complex operations with just a few elementary actions compared to a long sequence of actions required to achieve the same goal manually. A detailed evaluation of the usability improvements according to the aforementioned GOMS models is therefore not necessary. Instead, the feedback of the developers has been used to refine the provided functionalities over time.

5.2 Developed GUI Components

During the time of application in the considered scenarios, numerous views have been developed and some of these are described and illustrated in the following. While many have been specifically developed to visualize the internals of particular algorithms, some views are applicable in any scenario. They illustrate the complexity of the tools required to debug and monitor autonomous mobile robots. Since RoboFrame is currently the used platform in both investigated scenarios, the examples are limited to this middleware.

5.2.1 Measuring Execution Time and Frequency

In order to analyze the runtime behavior of the robot control software a view has been developed, which displays the execution time and frequency of each component (Figure 5.3). This enables monitoring the resource consumption of each algorithm. Several statistical data is included to identify bottlenecks, which might occur only for specific input data. It can be used in combination with software-in-the-loop testing as well as for remote debugging in order to improve and verify the runtime performance of particular algorithms.

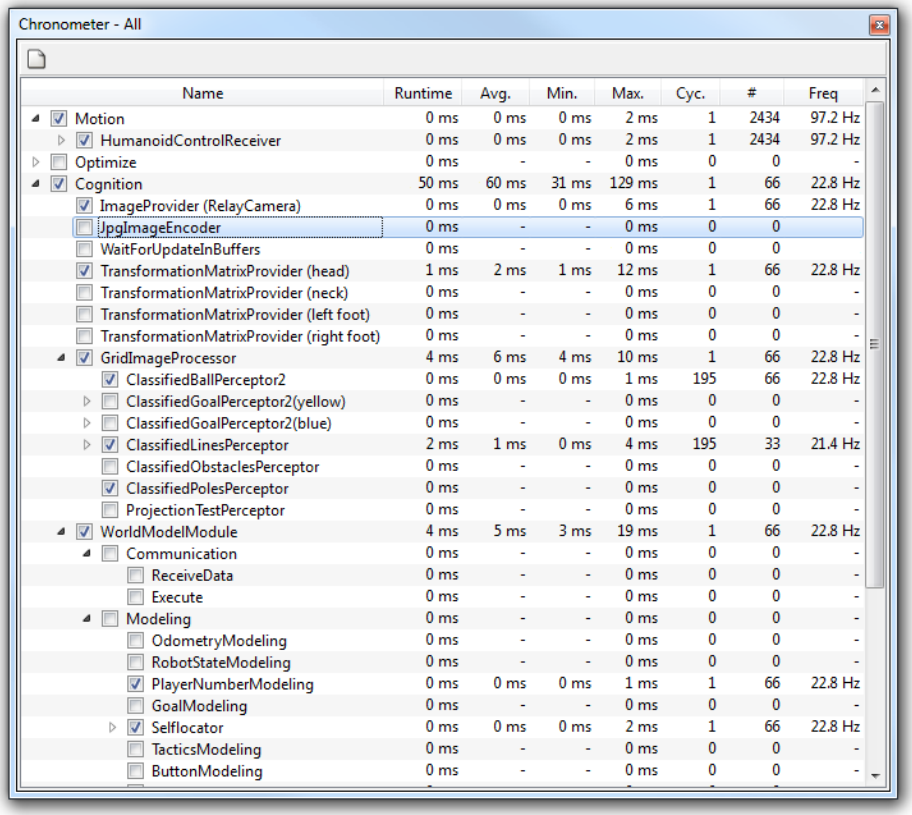


Figure 5.3: The view displays the execution time and frequency of each component of the robot control software

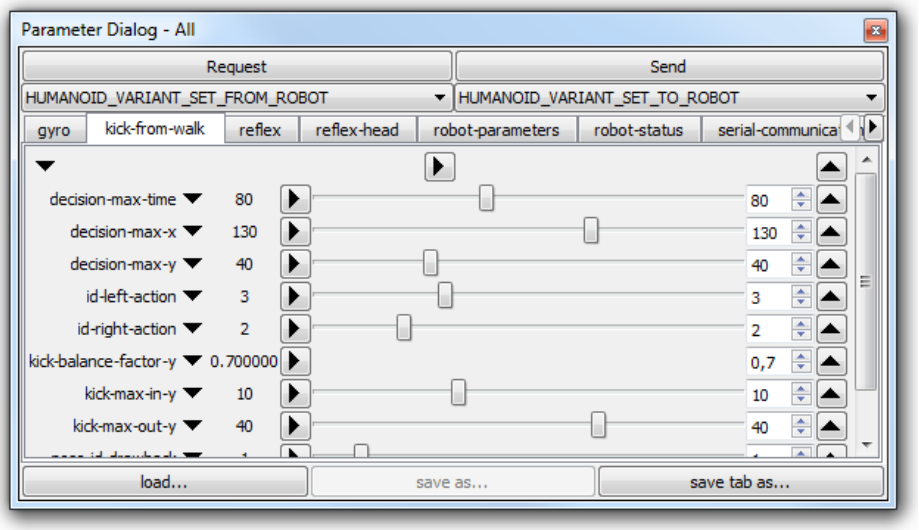


Figure 5.4: A generic view for the parametrization of arbitrary components at runtime

5.2.2 Generic Runtime Parametrization

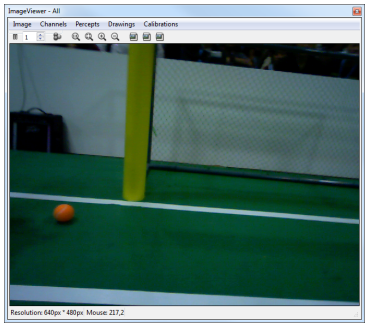
The various algorithms from different domains utilized in the control software of autonomous mobile robots are quite complex. Commonly, they are configurable by a large set of parameters, which need to be adjusted for satisfactory results. This parametrization needs to be done at runtime in order to determine suitable parameters efficiently. For this task a generic view has been developed. Each parameter set consists of key-value pairs, whereas the value can be of any primitive type. It can be utilized independent of the particular scenario, e.g., it is applied for configuring the trajectory generation and motion control algorithms of the biped robot as displayed in Figure 5.4.

5.2.3 Image Viewer and Image Processing Analysis

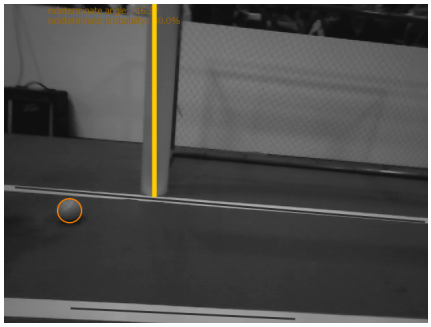
For the specific domain of image processing a view has been developed for displaying the camera images and debugging the image processing algorithms (Figure 5.5a). The detected objects are optionally visualized on top of the image, which can be drawn in gray-scale for better visibility of the overlays, as depicted in Figure 5.5b. Furthermore, additional graphical debug information of the detection algorithms can be shown, e.g., of the field lines detector as illustrated in Figure 5.5c, in order to analyze the intrinsics of the implementation.

5.3 Extension to Multiple Robots

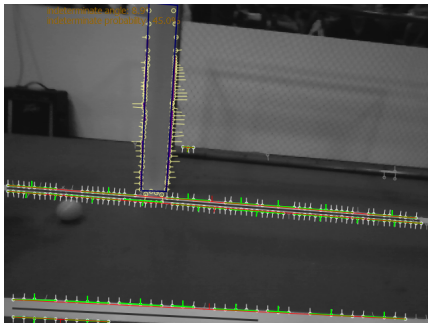
The aforementioned graphical components are a priori only capable of handling messages sent from a single source simultaneously. As soon as the GUI is connected with multiple robots, for each message bus multiple publishers exist. The graphical components receive messages from multiple robots and therefore the visualization of these data alternates, as only the latest information can be displayed. The same issue arises when the component sends messages back to the robot as these data are received by all agents.



(a) The view displaying the camera image



(b) Detected objects overlaying the gray-scale image



(c) Graphical debug information of the edge detection

Figure 5.5: The image viewer displays the raw images, visualizes the detected objects and enables graphical debugging of the algorithms' internals

Hence, the views must be aware of the different sources in order to support visualizing data from multiple robots at the same time. The received messages must be correlated with the particular robot and the information of multiple sources.

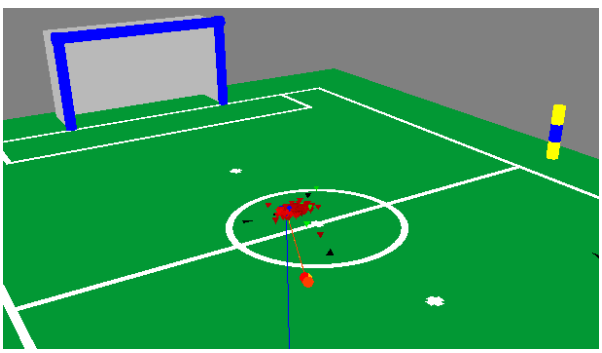
Notably, even when the GUI is connected with multiple robots simultaneously the data is never relayed from one robot through the graphical user interface to other robots.

5.3.1 Visualizing the World Models

A view has been developed to visualize the various different world models mentioned in Section 4.4. Each of the models is capable of drawing itself into a known map as illustrated in Figure 5.6. In the view used for the soccer scenario the particles of the monte-carlo method are drawn by the self locator model. Each hypothesis is indicated by a triangle, the color indicates the quality of the particle.

Likewise, the ball model provides the visualization of the ball. Here, too, the color is used as an indicator of the uncertainty of the model.

(a) Image of an external camera



(b) Visualization of the intrinsic data of one robot

Figure 5.6: The model viewer visualizes multiple different parts of the world model

To enable visualizing information from multiple robots simultaneously, the view has been enhanced to not only store each kind of message but also to keep separate storages for each source. Thus, for each robot all models can be visualized in a single view as shown later in Figure 6.11.

5.3.2 Reusing Single-Agent Debugging Tools

The majority of views for debugging and monitoring is focused on single robots. For example, displaying the camera images, changing parameters and introspecting the behavior decisions, to name just a few. These views are only applicable when communicating with a particular robot and cannot be used with multiple sources simultaneously. As mentioned in Section 3.3.5 the communication of these views needs therefore to be restricted to a single robot at a time.

In order to prevent altering all views, the concept as applied for throttling messages at the publisher, is applied in a modified manner. Instead of passing the subscription to all publishers and let these perform the throttling, it is only forwarded to a single robot. In RoboFrame the targets of a message sent over a particular message bus are further restricted as presented in Section 4.2.3. For ROS with its' multiple point-to-point connections a custom strategy at the subscriber node is applied, to utilize a single connection to a particular publisher only.

The correlation, which graphical component should communicate with which particular robot, is done using a central configuration view as depicted in Figure 5.7. This policy is enforced by the strategy implemented in the gateway which alters the information exchange between the graphical components and the utilized middleware. Therefore, the views receive only messages from a single robot as depicted in Figure 5.8, which shows multiple instances of the same view, each displaying the camera image of a particular robot.

For an increased usability the configuration cannot only be changed using the aforementioned view, but also using keyboard shortcuts. Thereby, the correlation of the currently selected view with a particular robot can be altered without switching to another view by using a single keystroke only.

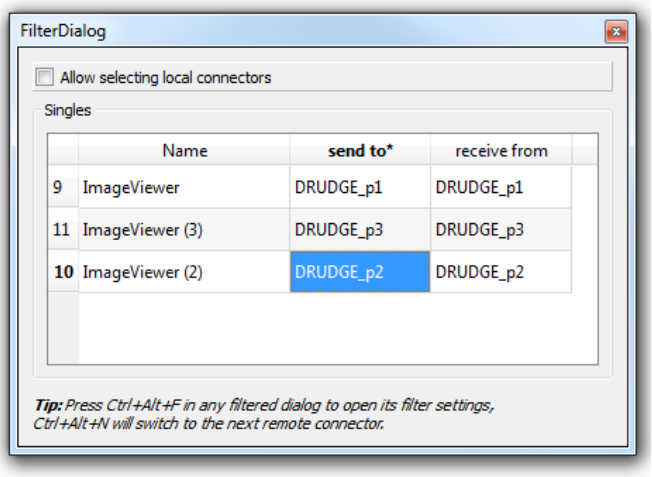


Figure 5.7: The central view for specifying the correlation between views and particular robots

Figure 5.8: Multiple instances of a single view, each communicating with a particular robot

5.4 Support for Testing, Debugging and Analysis

The aforementioned tools can be used for online debugging and monitoring. But for several tasks the subsequent analysis is more reasonable as stated in Section 3.1.4.

5.4.1 Recording Intrinsic Data

For comprehensive analysis tasks the recording of the intrinsic data of the control software is done directly on the robot as described in Section 4.3. Additionally, it is also possible to record the information in the GUI, either when the amount of data is manageable or a test is conducted with a wired connection featuring much more of bandwidth.

Therefore, a view has been developed for writing a dynamically configured set of messages to a logfile. The configuration is similar to the aforementioned view for remote logging (Figure 4.10) except that the selected messages are recorded in the GUI instance instead of locally on the robot.

Additionally, it is used for listing all messages included in the logfile (Figure 5.9). The recorded data can be played back at an arbitrary speed ranging from message-by-message to faster than real-time. It is utilized for various testing and debugging tasks involving repeated processing of the same input data.

Combining the aforementioned features of recording intrinsic data, reviewing them offline using the various different views and the applicability of these tools to teams of robots enable a comprehensive analysis not possible in most other approaches. One exemplary application illustrating these capabilities is described in Section 6.6.1.

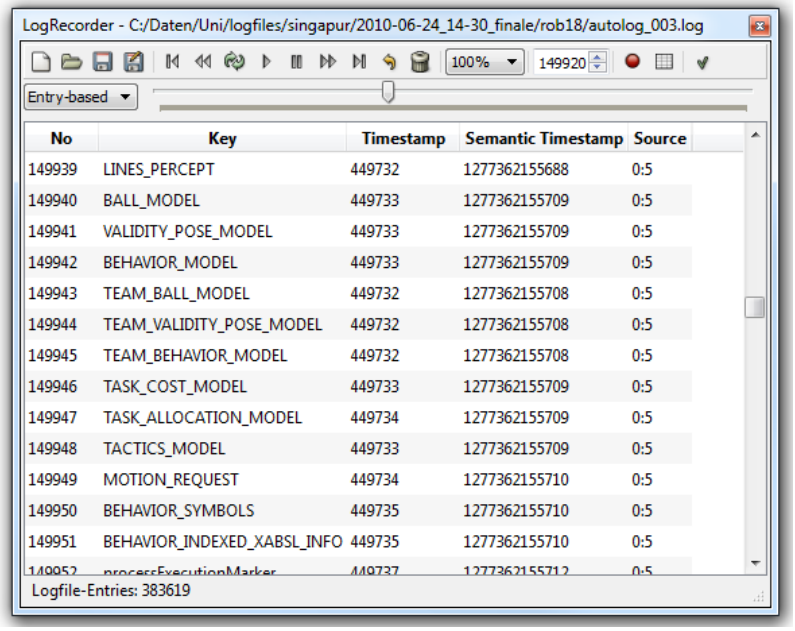


Figure 5.9: The view records arbitrary messages in the GUI and plays back these logfiles afterwards

5.4.2 Attaching External Data

Since the intrinsic data from either a single or even multiple robots may not be sufficient to gain a good overview of the surroundings during the analysis, the integration of additional external sensors is important.

Some approaches, which integrate external data, have been presented in Section 2.3.7. While these are limited to a subset of the robot control software, the developed method combines the advantages of the comprehensive debugging and monitoring infrastructure with additional external data. Thus, the analysis tasks are not limited to a specific component of the control software, but cover all aspects and are even applicable to teams of robots.

Therefore, single or multiple external video cameras are used to record the environment from external view points. This provides valuable information for the developers when analyzing and comparing the intrinsic data with the real situation.

The implementation is based on the widely used FFmpeg[2] library and therefore supports a widespread set of video codecs. In the described scenarios a general-purpose video camera was used to record the environment of the robots with a resolution of 720p in the MPEG-4 format.

Integration of External Videos

Besides the logfiles containing the intrinsic messages, the GUI has been enhanced to also play back video files of external cameras as described in [100]. In order to align the timing of the video with the automatically synchronized logfiles, a manual calibration of the offset between each video and the intrinsic data is required. This procedure is only necessary *once* for each video as the offsets are saved for subsequent usage. Thereafter, the videos are played back synchronously with the intrinsic data.

[2] http://ffmpeg.org

5.4.3 Event-Based Navigation of Data

The applications of the considered scenarios have gained additional functionality and integrated more complex algorithms over time. This results in an increasing amount of data exchanged between the components of the control software. For example, in both scenarios, during a ten minute mission approximately half a million messages are exchanged (Table 6.3).

The initially developed view for replaying the logged messages uses a message- and time-based navigation concept to browse through the recorded data. While this is applicable for replaying single messages over and over again for testing or cycling through a manageable amount of messages, it is not practical for browsing through the enormous quantity of messages of complex applications.

The user requires more useful information about the content of the recorded messages for efficient navigation. Therefore, interesting points in time need to be marked with custom events as suggested in Section 3.3.4. The kinds of events are arbitrary and can be either used for easing the navigation or identifying known problems, which require manual analysis.

The interesting points are detected using different algorithms, each focused on a particular type of event. Application specific algorithms can be easily integrated by registering the new classes at the developed infrastructure. The view enables the user to select the detection algorithms, which shall be executed for the loaded logfiles in order to analyze the messages automatically.

In Figure 5.10 a list of several hundreds events is shown, whereas the complete list of messages would have been thousand times longer.

Generation of Events based on Game States

In the soccer scenario the most usable occurrence for navigating through the logfiles of a soccer match is the course of the game. In this scenario the current game state is continuously broadcast in the wireless network using UDP datagrams, which simplifies the detection of changes significantly.

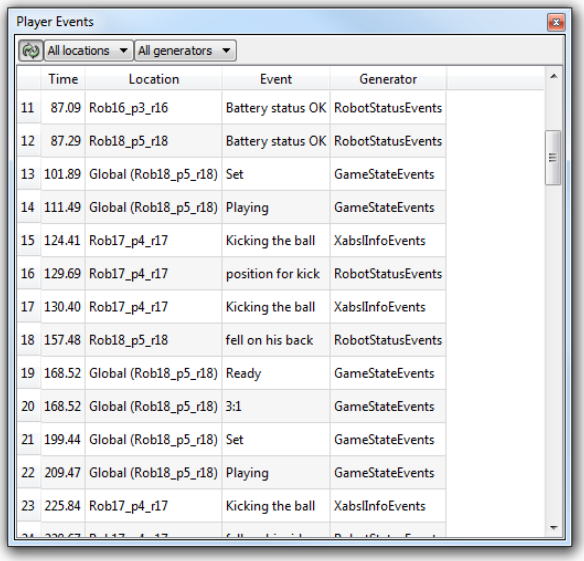


Figure 5.10: List of detected events easing navigating through the numerous amount of recorded data

The game state includes various different information, like the playing time, the current score, penalties for each player and the current phase of the game.

Each robot records these messages to the local logfile. The detection algorithm cycles through all recorded messages and generates appropriate events, when the game state changes. In Figure 5.10 the list of game state events extracted from a normal soccer game is shown alongside other event types. Based on this information, the navigation through the recorded messages is significantly improved.

As the game states are recorded by any of the teammates, the algorithm merges similar events, which are temporally close together.

Detected Victims and Hazardous Objects

An incident specific to the rescue scenario is the detection of victims and hazardous objects. Analogous to the game state in the soccer scenario, the detected objects are recorded in the logfile. These messages are then extracted and corresponding events are generated in order to ease locating the corresponding points in time.

5.4.4 Automated Analysis with Custom Event Generation

Besides these events, which are mostly only used for a better overview of the recorded data, additional events are generated to identify known problems automatically. Such information permits the user to focus on relevant periods and efficiently review the recorded data, as these events identify situations which are most relevant for manual analysis.

Several different detection algorithms have been implemented to make the analysis tasks of the selected scenarios more efficient as described in the following.

Events when a Robot Falls Over

For a biped robot a specific incident is the falling during locomotion. Based on the recorded intrinsic information including the data of the robot's accelerometers, the issue of falling over can be detected. Besides the point in time of falling, the duration until the robot was able to stand up again is also of interest. Therefore, for both points in time an event is generated.

Discontinuity of the Self Localization

The task of self localization for mobile robots is to determine the current position and orientation of the robot. As long as the mobile platform is not manually repositioned, it can only move in space in a continuous manner and with a limited speed. Therefore, if the position of the robot changes considerably in a short timeframe, the determined location must either have been wrong before or afterward. An algorithm detects such discontinuities using predefined thresholds and generates events at these points in time.

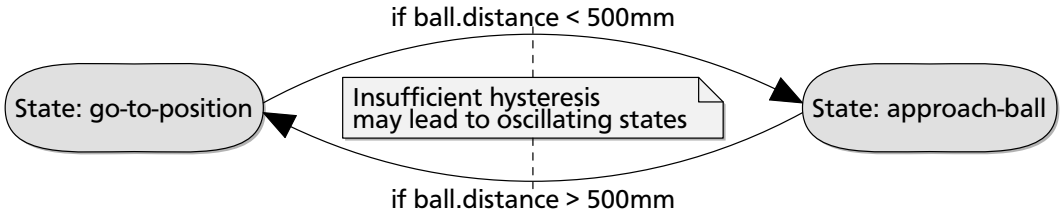


Figure 5.11: Conditions of the state transitions with insufficient hysteresis lead to oscillating states in the behavior control

State Oscillation in the Behavior Control

Another example, which shows the gain of the automated analysis, is from the domain of behavior control. In the considered scenarios the behavior is implemented using hierarchies of finite state machines defined in XABSL [69]. A common problem with this approach exists when the state machine continuously oscillates between multiple states. This is usually due to insufficient hysteresis of the transition constraints as illustrated in Figure 5.11, which is especially valid for robotics due to the numerous uncertainties.

An algorithm has been implemented to detect such occurrences in the behavior and generates corresponding events. These problems are especially difficult to find manually as the timeframe of the occurrence is very short. Thus, the time to identify these cases is reduced significantly and users can concentrate on tracking down the source of the issues.

5.4.5 Augmenting External Video with Intrinsic Information for Manual Review

The individual views are particularly suitable when the information is debugged and analyzed. In order to obtain an overall picture of the mission, the manual review process requires a single integrated view merging the most important information. As the visual perception of humans is especially well-marked, a visual representation has been chosen. The developed view displays the external video augmented with the automatically extracted events as depicted in Figure 5.12. The various different types of events can be flexibly arranged and individually enabled and disabled depending on the particular tasks. Based on this information the developers can efficiently obtain an overview, after which particular points of interest can be analyzed in detail.

5.4.6 Automated Comparison of Competitive Algorithms

In the robotics domain various different algorithms are used to achieve similar functionality, e.g., for object recognition or self localization. Furthermore, most implementations require numerous parameters to tune the procedure to a particular environment. Comparing these competitive algorithms or differently parametrized instances of the same implementation is challenging. When conducted manually, it is a very time-consuming task and the results are rather subjective. In order to achieve objective results efficiently, this process has been automated.

The algorithms being compared offline can utilize any messages recorded. The set of input data and the computed results vary based on the demands of the specific tasks. For example, when comparing image processing implementations the raw image as well as the transformation matrix

of the camera is stored together on a per-frame base. Likewise, all detected objects are a single frame are grouped as the result for each frame for later comparison.

The developed view enables selecting the logfile containing the necessary input data as well as two or more competing algorithms. Each algorithm needs to process all input data and provides the corresponding results for later comparison. Existing components of the robot control software can be used without any modification. The job control is responsible for feeding all input data to each algorithm sequentially or for parallelizing the execution of algorithms in order to efficiently utilize modern multi-core processors to capacity.

After the algorithms have processed the data the results are evaluated, visualized and automatically compared. While the comparison must be implemented for each application, the visualization can reuse already developed widgets. For example, the comparison between the different results of a single frame can be visualized as a simple traffic light. Green indicates that the results of both algorithms are equal, yellow that they are similar and red illustrates diverse data as shown later in Figure 6.15. Also, more complex visualizations involving all data have been developed, like graphs displaying the results of multiple algorithms for all processed frames in a joint view (Figure 6.16).

The efforts required to utilize the developed comparison infrastructure are reduced to a minimum. A particular application where this tool has been applied is described in detail in Section 6.6.4. For future analysis and comparisons only a custom storage class, which holds the input data as well as the results of the algorithms, and a comparison function need to be implemented.

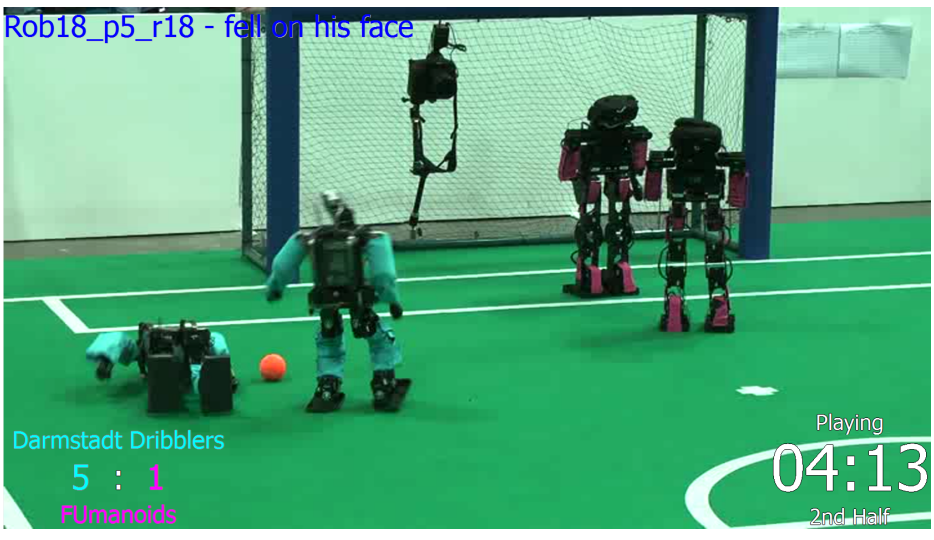


Figure 5.12: The external video is augmented with intrinsic information of the multiple robots

6 Applications and Results

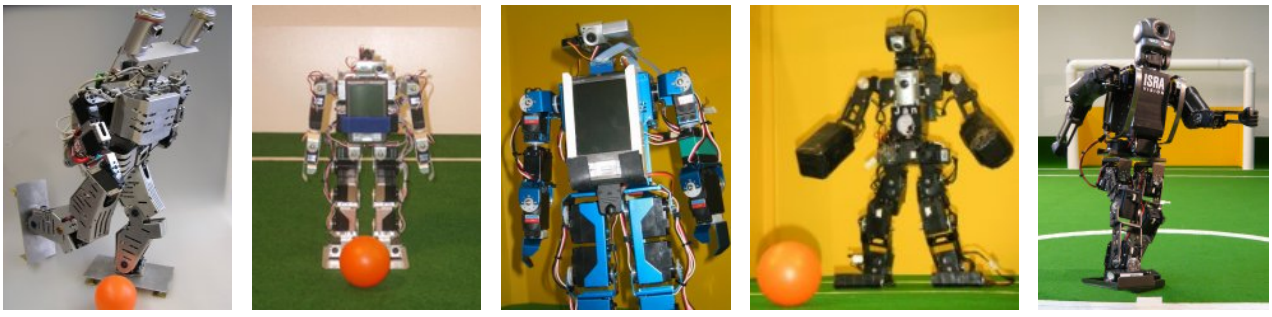
This chapter starts with an overview of applications where the developed software has successfully been applied. Section 6.2 presents the benchmark results for local message exchange proving the significant performance improvements achieved with the developed concepts. In addition, the influence of local recording of messages on runtime performance is stated in Section 6.3.

In Section 6.4 the applications for explicit inter-robot communication are described. On the one hand, the exchanged information of the team model is considered as well as the observation of this data during the mission. On the other hand, the application of dynamic role assignment is detailed, which enabled a cooperative team behavior using the developed communication mechanisms. Finally the utilization of the integrated GUI is shown by reference to some example scenarios in Section 6.5 and the successful applications of the developed tools for automated analysis are shown in Section 6.6.

6.1 Applications

6.1.1 Humanoid Robots

RoboFrame has been utilized in the humanoid soccer scenario by the Darmstadt Dribblers since 2004. Within this period the hardware has evolved significantly and the software had to be adapted to the heterogeneous hardware (Figure 6.1). Likewise, many algorithms have been developed on top of RoboFrame to provide sophisticated functionalities and tools.



(a) (b) (c) (d) (e)

Figure 6.1: Different humanoid robots participating at RoboCup:
(a) Mr. DD (2004), (b) Mr. DD Junior (2005), (c) Mr. DD Junior 2 (2005), (d) DD2007 (2006-2008), (e) DD2008 (since 2008)

6.1.2 Rescue Robots

The Team Hector has been established in 2008 and competed for the first time in the Robot Rescue League at RoboCup 2009. As it utilizes RoboFrame many existing components and tools for debugging and monitoring originally developed for the soccer team have been reused. This ranges

Table 6.1: Successful works performed on top of RoboFrame

Kind of work	Quantity
Internships	6
Bachelor theses	3
Master theses	11
Journal papers	2
Conference papers	7
Workshop papers	4

from fundamentals like drivers for the hardware and kinematic models to high-level functionality like interfacing with simulation and integration of a behavior language. The generic infrastructure for remote monitoring, recording of intrinsic data and offline analysis was likewise available.

The hardware of the wheeled vehicle consists of two computers. With the utilization of RoboFrame the computational load is transparently distributed across these machines.

Additionally, active sensors, which are not allowed in the soccer league, have been integrated. Based on these sensors further developments in the domain of mapping, self localization and path planing have been conducted.

Due to the high degree of reuse and the proven software platform the team was able to quickly succeed in the new league.

6.1.3 Employment in Teaching

Besides the RoboCup teams, RoboFrame has been actively used in teaching at the author's group. A large number of student theses and practical courses have been based on RoboFrame (Table 6.1). This demonstrates the low effort necessary to utilize the software, so that developers can focus on their specific tasks in the robotics domain.

Furthermore, the developed software has been utilized on other hardware platforms in the context of teaching, in particular on a Pioneer 2 DX and a newly developed four-legged robot [37, 38] (Figure 6.2).

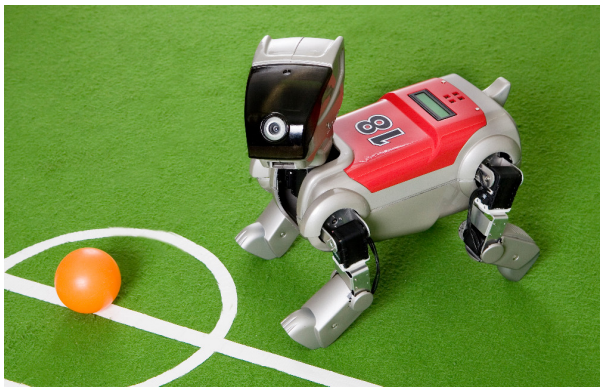


Figure 6.2: The newly developed four-legged robot

6.1.4 Application in Other Institutes

TU Delft, TU Eindhoven, University of Twente

The three universities of technology in the Netherlands have joined their efforts in the *Dutch Robotics* project. Their long term vision is to develop a new generation of robots. The team has been competing in RoboCup Soccer since 2008. Their new robot *TUlip* [47], which is currently under development, is utilizing RoboFrame and will participate at the next competition in 2011.

Various Groups at TU Darmstadt

The presented rescue scenario founded by the Research Training Group (GRK 1362) consists of several inter-disciplinary groups at TU Darmstadt. The members from Simulation, Systems Optimization and Robotics, Flight Systems and Control Theory, Multimodal Interactive Systems as well as Image Understanding are using RoboFrame for integrating the domain specific algorithms for the wheeled vehicle [73].

TU München

The Institute of Cognitive Systems at TU München participating in the Cluster of Excellence Cognition for Technical Systems is evaluating different middleware for their application scenarios and considers the use of RoboFrame.

6.2 Local Message Performance

6.2.1 Messages in Considered Scenarios

The types of messages exchanged in the considered robot control software are very divers. The size varies significantly as well as the number of messages. In Figure 6.3 the quantity of messages exchanged between the components is illustrated for both aforementioned scenarios.

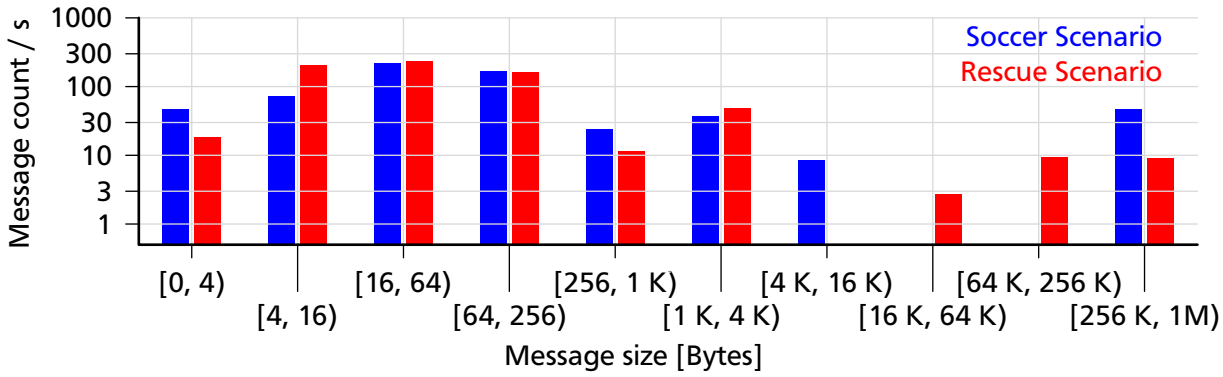


Figure 6.3: The number of messages exchanged per second inside the robot control software

Table 6.2: The differentiation of small and large messages exchanged in the scenarios

Scenario	All messages Byte/s	Messages < 64 Kb Byte/s	Messages >= 64 Kb Byte/s
Soccer	23.7 M	0.19 M	23.51 M
Rescue	7.9 M	0.2 M	7.7 M

These measurements have been conducted during normal operation and have then been normalized to the period of one second. The quantity of small messages is significant. But also notable are the few quite large messages which equate to the raw images passed from the camera image acquisition to the image processing component. Despite some differences between both applications the characteristic of having numerous small messages and few large ones is common for the complex control software of autonomous robots. The amount of data in total as well as the proportion of the large raw sensor data is shown in Table 6.2.

6.2.2 Measurement of Latency

The latency of the message exchange has been measured as stated in Section 3.2.1. While the presented benchmark measures the round trip time of a message, the following graphs always show the latency of a unidirectional communicated message. The measurements of the warm-up and cool-down phase are not shown in the illustrations as they must not be considered for the statistics. While numerous of runs have been conducted a single representative has been selected for illustration.

All subsequent benchmarks have been performed on an Intel Atom N270 running at a fixed frequency of 1.6 GHz. This CPU has replaced the former AMD Geode 800 running at 500 MHz used in the soccer scenario until beginning of 2010. But since the AMD Geode does not support a high precision event timer which is used for accurately measuring the latencies, the tests have been executed on the modern CPU. The system is based on a Linux operating system, namely Ubuntu 10.04 Server, with a generic kernel (version 2.6.32-24) and was idle during the test.

In Figure 6.4 the measured latencies for exchanged messages with a fixed-size of 2 Kilobytes using RoboFrame are depicted. Each measurement is marked with a cross, the average is shown by a green line and the standard deviation with red lines. Each message consists of a single data block, which does not require costly marshaling. The impact of marshaling complex objects is not explored further as it is not relevant for the improved message exchange using references.

The latency varies between 0.06 ms and 0.1 ms and the average is at 0.067ms. These variances are due to the scheduling of the operating system and concurrent tasks inherently executed in parallel to the benchmark.

Real-Time Kernel

When the same test is performed on an operating system with a real-time preempt kernel (version 2.6.31-11-rt), the latency varies much less (Figure 6.5). But the average latency using the preempt kernel is notably higher than on the generic kernel, when the system is idle. The measurements range from 0.095 ms to 0.11 ms. The real-time (RT) improvements come at the cost of a possible performance degradation.

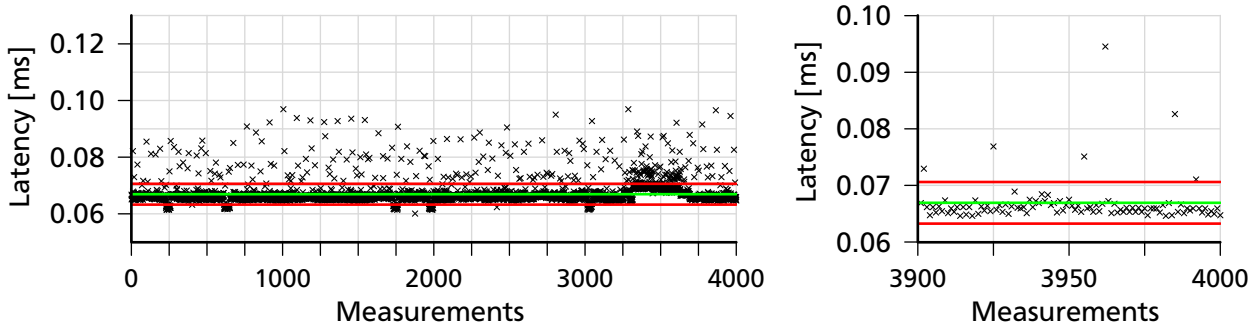


Figure 6.4: The measured latency for local message exchange on a generic kernel, (with a message size of 2 Kb using RoboFrame)
The average is shown in green and the standard deviation is shown in red

6.2.3 Impact of a Real-Time Kernel under System Load

The advantages of a real-time preempt kernel emerge when the system is not idle but under high load. To show this, in the following measurements the CPU was put under a synthetic load of 99 % using the software *lookbusy*[1]. The test is started using the FIFO scheduler and, as a result, the benchmark program is not preempted by other processes.

In Figure 6.6 the measurements of the latencies are compared for generic and preempt kernels, each under load and idle. For better visualization the data is plotted as box-and-whisker diagrams [103] which show the spread of the data. The median is depicted with a black line and the green box ranges from the 25^{th} to the 75^{th} percentile covering the interquartile range. The whiskers are drawn at the 2.5^{th} and 97.5^{th} percentile which span a range of 95% of the measurements. The measurements not included between the whiskers are displayed as red crosses.

The first two columns display the measurements of the previous Figures 6.4 and 6.5. The ratio between the range of the boxes and the whiskers is significantly different for the two considered

[1] http://www.devin.com/lookbusy/

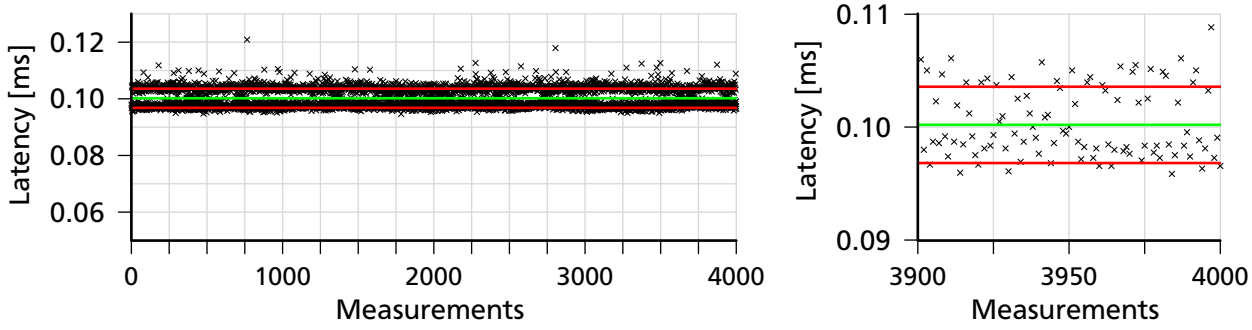


Figure 6.5: The measured latency for local message exchange on a system using a real-time preempt kernel, (with a message size of 2 Kb using RoboFrame)

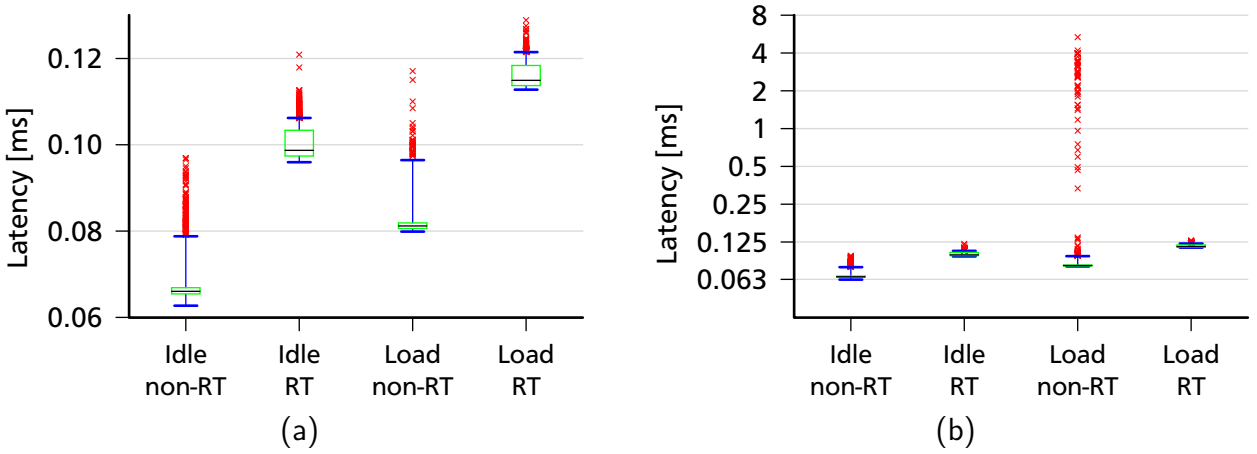


Figure 6.6: Comparison of latency for local message exchange with and without a real-time kernel while the system is idle or under load, while (a) and (b) plot the same data but with different scales (with a message size of 2 Kb using RoboFrame)

kernels. Especially under load the increased latencies of the non-preempt kernel are notable as shown in Figure 6.6b. The extreme outliers are about 40 times higher than the average latency. In contrast, the latency measured on the system using the preempt kernel never exceeds 0.13 ms, which is only ten percent higher than the average latency.

This effect has been stated in [87], which promotes utilizing real-time kernels for robotic systems even without running the software itself in real-time.

Latency for Different Message Sizes

The latency of the message exchange between local components depends highly on the size of the messages. Therefore, the latencies for different message sizes used by the two scenarios have been measured. The test has been implemented and conducted for several different existing middleware, namely ACE, ICE, Player, RoboFrame and ROS. Although the approaches vary in details the benchmark has been realized as uniformly as possible.

Some changes required to perform the tests for all middleware are noted briefly.

- *ICE* — restricts the message size to a default size of 1 Mb. For the test with the largest message this limits needs to be increased due to several bytes of protocol overhead. The configuration option *Ice.MessageSizeMax* has therefore been increased to 2 Mb.
- *Player* — to enable round trip testing a driver and client is used for the benchmark. To speed up the conduction of the evaluation the default rate of 100 Hz of the Player server has been increased to 10.000 Hz.
- The other approaches have been used without any modifications or special parameters.

The average latency for each message size is presented for the evaluated middleware (Figure 6.7). Due to the differences and specific available options of each approach an experienced used might be able to tweak the results. But the trend of all evaluated middleware is similar even if the

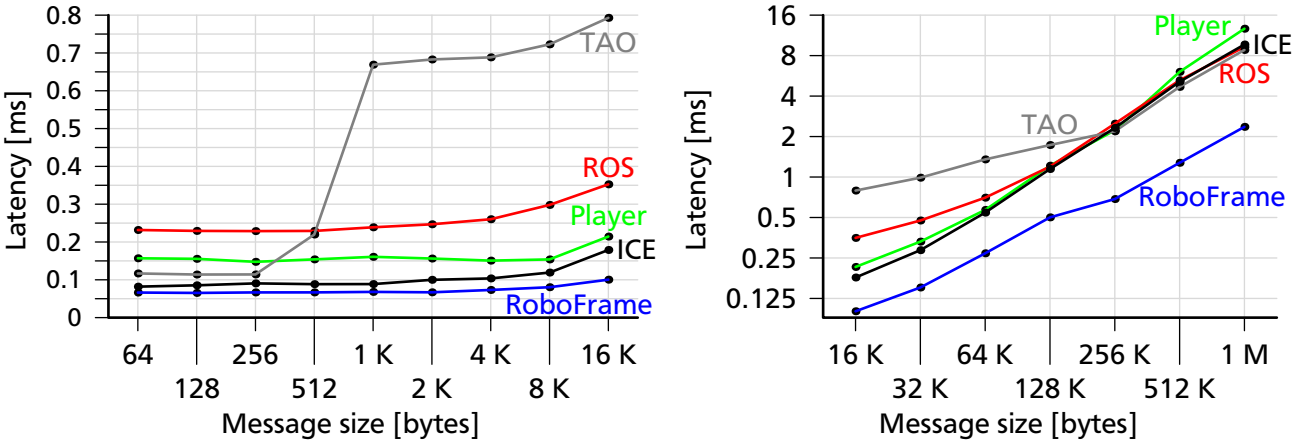


Figure 6.7: The latency for exchanging messages with different sizes locally using various middleware (note the different scales on the y-axes)

concrete latency is higher or lower. Especially, the difference in magnitude for large messages is substantial. The different performance of the four middleware can be explained by their respective communication concepts.

In RoboFrame the components are statically compiled into a single binary and therefore avoids the IP stack for the message exchanged between those. It only requires marshaling, memory copy and demarshaling of the data and therefore requires less resources resulting in a lower latency compared to the other approaches.

All other approaches utilize the IP stack for the data exchange between the locally executed components. The significant impact is especially visible for larger sizes. This is due to the fragmentation performed by the IP stack which occurs commonly at a size of 8 Kb (even if the IP protocol permits a size of up to 64 Kb). Even if the latency of the different middleware differ for smaller messages, they all converge for larger sized data.

However, a latency of several milliseconds for larger messages (e.g. at an approximate rate of 30 Hz for images), which is caused by all of the tested middleware, poses a *significant overhead* which is especially detrimental for the restricted platforms in the considered scenarios.

6.2.4 Impact of Reference Passing

The latency scales almost linearly for larger messages sizes. The major impact is put down to memory copy operations as long as the IP stack is not involved. Therefore, according to the presented concept messages are passed by reference instead. Consequently, the overhead due to marshaling, memory copy and demarshaling vanishes. This reduces the overhead of local message exchange to a minimum, which is constant, as it is independent of the size of the message, as depicted in Figure 6.8.

Due to the elementariness of the reference passing the latency remains extremely low as shown in Figure 6.9 even if the system is under load as described earlier. The results are equal for all different messages sizes. Again, the usage of a preempt kernel avoids the outliers in exchange for an increased average latency.

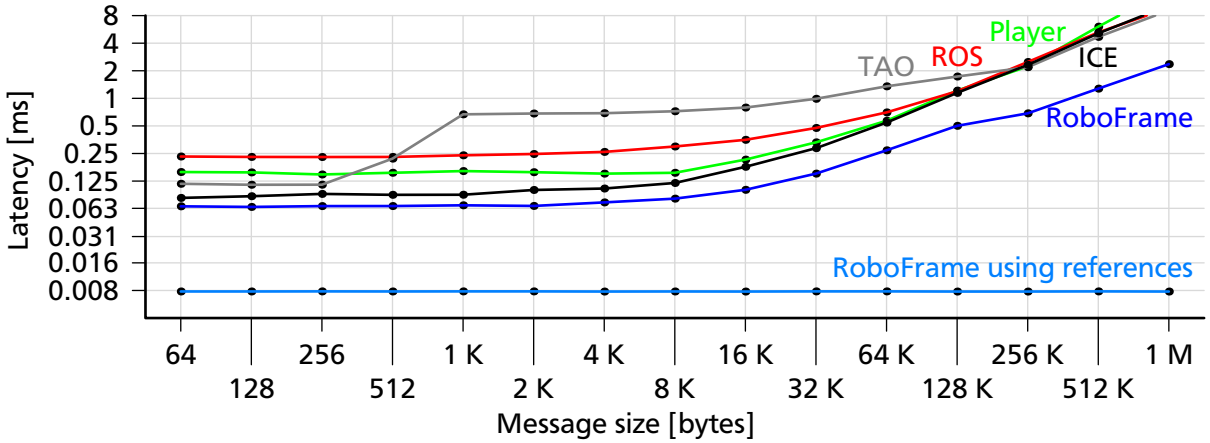


Figure 6.8: The latency for different message sizes is reduced to a minimum when utilizing pass-by-reference in RoboFrame

Improved Efficiency

The concept of passing messages by reference is utilized in both scenarios. All components, which do not necessarily need to run concurrently, are executed sequentially to enable exchanging messages by reference.

On the humanoid robot, powered by a single core processor, all relevant components are combined in two threads. All components except the motion control are running sequentially at a rate of 30 Hz. For the wheeled mainly the components for sensor data acquisition and processing are aggregated.

The impact on the number of messages and bytes, which still need to be marshaled, memory copied and demarshaled is depicted in Table 6.3. In the soccer scenario the number of messages is reduced significantly, since nearly all messages are passed by reference. For the rescue robot the fraction of messages passed by reference is significantly lower, since most of the smaller messages are transferred between the two computers or different threads on the same host. Only a small number of the messages is thereby passed by reference.

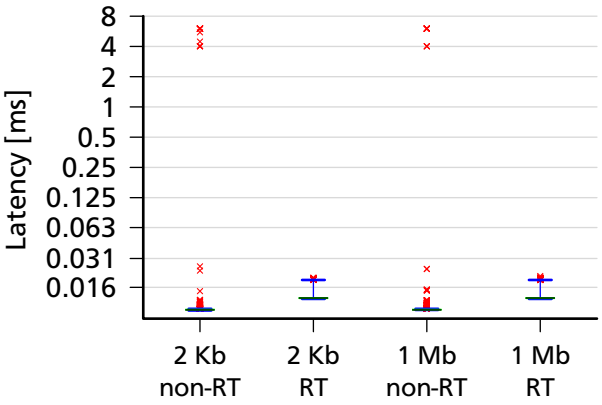


Figure 6.9: The latency for passing references in RoboFrame with and without a real-time kernel while the system is under load

Table 6.3: Reduced overhead for local message exchange

	All messages		Messages copied		Messages passed by reference			
Scenario	Msg/s	Byte/s	Msg/s	Byte/s	Msg/s		Byte/s	
Soccer	616.1	23.7 M	88.3	0.004 M	527.8	85.7 %	23.694 M	99.98 %
Rescue	693.9	7.9 M	658.7	0.357 M	35.2	5.1 %	7.543 M	95.48 %

But for both scenarios the *amount of bytes copied* is *drastically reduced*. This is, because especially the messages with the largest sizes are passed by reference wherever possible. The reduced demand for resources by the middleware permits the functional components to utilize more resources in order to improve the overall performance of the robot.

6.3 Recording of Messages Locally on the Robot

The recording of messages locally on the robot is the mandatory prerequisite for any offline analysis. For a complete review, all messages would need to be recorded. But the amount of data recorded per second would exceed the restricted bandwidth of the local storage.

While the recording is only limited by the storage bandwidth for the current humanoid robot consisting of an Intel Atom processor, on the former platform based on an AMD Geode the logging was additionally bounded by the computational resources since an increased amount of recorded data directly reduced the achievable frame rate of the image processing component.

Feasible Subsets of Messages

Due to the limitations of the mobile platforms only a subset of messages can be recorded which poses a trade-off between completeness and resource-efficiency. The numerous small messages exchanged between the image processing and the world model as well as between world model and behavior control are recorded for every frame as they are crucial for a detailed analysis and altogether still small in size as aforementioned in Table 6.2.

However, the raw images cannot be recorded for every frame. As the compression using the JPG algorithm poses an additional overhead, only a fraction of the raw images is recorded without any compression applied. The utilization of the developed efficient publisher-side throttling is mandatory in order to avoid any extra overhead for skipped messages. Exchanging the messages by reference is not feasible since the writing to the storage must be performed concurrently in order not to block the rest of the robot control software.

Depending on the number of images actually recorded, the performance of the system degrades as stated in Table 6.4. These measurements have been taken while the humanoid robot was standing still in order to avoid differences due to variations in the test situation.

During all test runs and missions the number of images logged was therefore limited to once every five seconds. This data is sufficient to determine problems related to the camera, e.g., damaged mounting of the camera, wrongly calibrated focus, inaccurate color calibration and influence of changed lighting conditions, while still keeping the impact of the recording on the overall performance reasonable low.

Table 6.4: Performance impact of recording large messages in the soccer scenario measured on an AMD Geode

Recorded Messages	Frame rate	Processed images per minute	Effective data rate
None	16.43 Hz	986	-
All percepts and models	16.32 Hz	979	142 Kb/s
Additionally one image every five seconds	15.80 Hz	948	262 Kb/s
Additionally one image every second	13.07 Hz	784	742 Kb/s

6.4 Team Communication

The team communication is currently only applied in the soccer scenario. But for the upcoming competition in the rescue scenario a team of two vehicles will cooperate in the tasks of exploration, mapping and victim detection.

The developed mechanisms provide explicit team communication between multiple robots. The size of the team can be adapted dynamically at runtime. Different communication paradigms can be utilized to suit the different conditions of testing and applications with real robots involved.

6.4.1 Communication on Various Levels

The team communication is used on various different levels.

On the level of the world model different information is shared between the teammates. Each robot communicates its own position and orientation as well as state information, e.g., if it has fallen. Furthermore, the relative ball position and speed are passed to the teammates.

This data is utilized by a decentralized dynamic role assignment to determine the role of the robot. The actual decision can also be communicated in order to prevent assigning the same role to multiple robots at the same time. Each role implies executing a custom behavior, each with different objectives.

An example of the dynamic role assignment is depicted in Figure 6.10.

On the behavior level the team communication is utilized to pass information about strategic decisions or events in the game. For example, when the robot is performing a kick the teammates are notified, so that they can track the ball and not look elsewhere, which might result in loosing the ball's location.

All information from the different world models, the dynamic role assignment as well as the internals of the behavior can be visualized using the developed GUI, in order to permit thorough testing and debugging of each component.

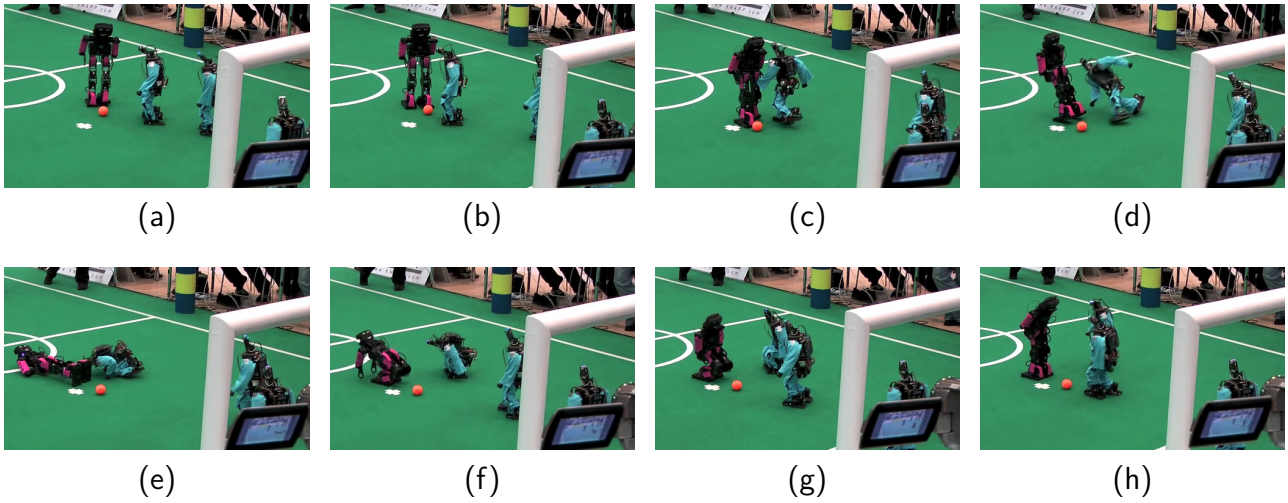


Figure 6.10: The dynamic role assignment is based on explicit team communication. In (a) to (c) the left robot with the cyan jersey is approaching and kicking the ball. His teammate is keeping distance in order not to interfere. In (d) the first robot is falling, communicating the incident to the teammates. Consequently, in (e) to (g) the second robot approaches the ball, even though the first robot is still located closer. Until the first robot has stood up again the second has already reached the ball as shown in (h).

The developed team communication enables sophisticated cooperative behaviors and the team of the Darmstadt Dribblers is one of the very few teams which showed complex team behavior at the competitions in 2009 and 2010.

6.4.2 Online Observation During Competition

The rules of the humanoid league prohibit any interaction with the robots during a soccer game. But it is allowed to listen to the wireless communication, which can be used to passively observe the exchanged information of the team communication.

In situations, when a team of robots does not perform as expected, it is often difficult if not impossible to determine the source of the problem from the outside. The intrinsic data of the team communication provides valuable information, which enables isolating the origin of a problem. This functionality has been actively used in the recent competitions in order to:

- verify wireless connectivity,
- examine the result of the self localization,
- monitor the model of the ball's position and movement and
- check the dynamically assigned role.

For example, when a robot has fallen on the camera it might effect the mechanical structure. A minimal bending of the camera mount results in severely wrong projections in the image processing. Such issues can easily be detected from the outside using this online observation tool visualizing the intrinsic models.

6.5 Integrated User Interface

The integrated user interface is the foundation of all developed tools for debugging and monitoring. The provided features ease the customization of the interface to the particular demands of each task. The improved usability permits the developers to focus on the actual job and puts the configuration of the tools in the background.

6.5.1 Debugging and Monitoring Tools

Numerous tools have been developed for the described scenarios. Various of these have been described and depicted (Figures 5.3, 5.4, 5.5 and 5.6). For every component in the robot control software the GUI features at least a single if not multiple different views for debugging and monitoring the internals of the algorithms. These tools can be applied online as well as offline based on recorded messages. Therefore, every single aspect of the application can be debugged and monitored thoroughly. Especially, when applied online over a restricted wireless connection, the developed throttling of messages at the publisher side is crucial in order to reduce the necessary bandwidth.

For example, the measurement of the runtime of each component is continuously applied in the considered scenarios. It has been successfully utilized to detect various situations where the runtime of a component was significantly increased due to very specific input data.

Playback of Recorded Data to the Control Software

The capability to feed the recorded messages back into a customized control software permits various debugging scenarios. It is applied to debug the algorithms and profile the performance of the implementation with known input data in order to obtain reproducible results which is not possible when performed online in a dynamic scenario. Especially, the usage of a debugger and profiler negatively affects the runtime behavior of the robot control software which might render the results useless.

For example, after observing a significantly increased runtime of the lines detection algorithm, the control software was fed with the recorded input data and the application was profiled. This enabled the developers to track down the source of the problem and improve the implementation for handling the identified special cases more efficiently. Afterwards, the procedure was applied again in order to validate the achieved improvements.

Software-in-the-Loop Testing

The simulation is actively used in the development process to verify the proper functionality of the robot control software before performing tests on the real robot. Besides the reduced wear of hardware, the integration with the simulation permits multi-level testing and the utilization of ground-truth data which is not available otherwise.

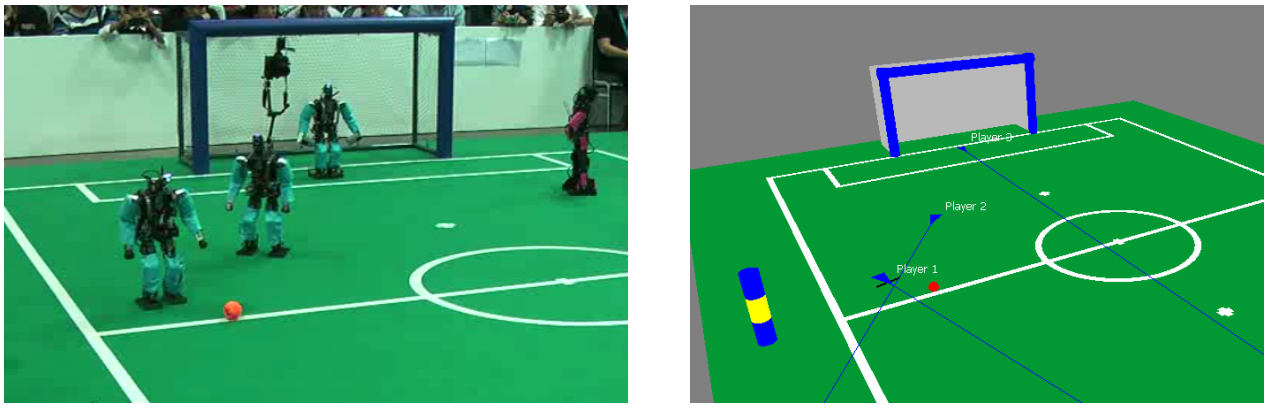
For example, software-in-the-loop testing is applied for the task of simultaneous localization and mapping in the rescue scenario. The uncertainties of the noisy sensor data as well as the influences of slipping and friction of the locomotion are avoided in order to focus on the high-level

functionality of the modeling and path planning. The simulation can provide ground-truth data of the robots' position and visible objects instead of processing the simulated sensor data and determining the information in the control software. For both scenarios the software-in-the-loop testing is applied to reduce the complexity of the robot control software involved and therefore eases the conduction of debugging tasks.

6.5.2 Extension to Multiple Robots

The developed GUI is not limited to work with a single robot but can communicate with multiple robots simultaneously. Each of the presented views can be reused in scenarios with multiple agents without the need for any modification. This has been achieved based on the presented hooks between the components and the middleware enabling single views to be restricted to communicating with a particular robot only. An example showing multiple instances of the same view, displaying data from different robots, is shown in Figure 5.8.

Furthermore, it also permits visualizing information from a team of collaborating robots in a single view. This application is illustrated in Figure 6.11.



(a) Image of an external camera

(b) Visualization of the intrinsic data from multiple robots

Figure 6.11: The view shown in (b) stores communicated information between teammates separately and can therefore visualizes data from a team of robots simultaneously

6.6 Sophisticated Analysis Tools

Based on the aforementioned infrastructure, a set of advanced analysis tools has been developed to make the debugging and monitoring of complex applications more efficient and result in repeatable, objective results. Some exemplary applications are described and illustrated in the following and the achieved improvements due to the newly developed software are stated.

6.6.1 Debugging Teams of Robots

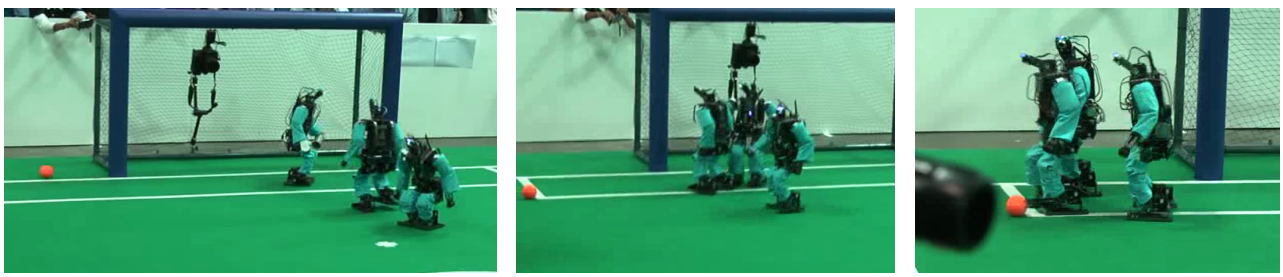
The intrinsic information of each robot control software is recorded locally on each robot. The presented throttling concept is crucial for performing this logging efficiently. The time offset between the multiple involved hosts is determined automatically, which permits the later synchronization without any manual intervention.

Additionally, external information from a video camera capturing the environment is supplied. Both, the intrinsic messages of multiple robots and the external video, are played back synchronously. Any of the aforementioned views can be utilized to debug and monitor the control software of each single agent and the external video enables the developers to evaluate the correctness or quality of the calculated results.

The recording of intrinsic and external data and the subsequent analysis is performed for every test run and mission. It enables in-depth debugging and monitoring and thus the detection of problems which are not even noticeable when observing the robots from the outside as well as the identification of the source of an issue.

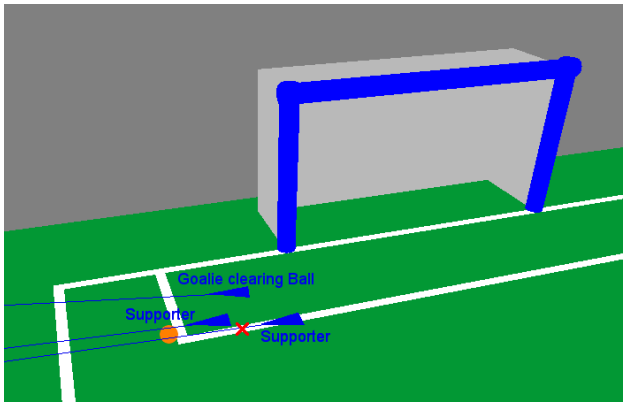
For example, in the situation depicted in Figure 6.12, at the first glance all robots seem to be approaching the ball obstructing each other. The initial assumption of the developers was either a defect of the dynamic role assignment, which should avoid multiple robots approaching the ball concurrently, or a problem with the team communication, whereby each robot would perform as if it was the only agent on the field.

Based on the comprehensive information available for debugging the behavior, it was possible to identify the reason of the problem. Both, the team communication and the dynamic role assignment, worked flawlessly which is attested by the current role of each robot as illustrated in Figure 6.13a. Instead, the target positions of the supporting robots were set to unreasonable coordinates near their own penalty area as shown in Figure 6.13b and indicated in (a) with a red

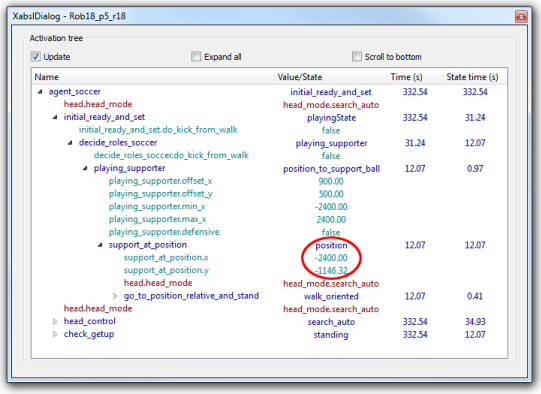


(a) (b) (c)

Figure 6.12: Difficulty to identify the source of problems in complex applications. All three robots seem to approach the ball obstructing each other in (b) and (c)



(a) The robots' poses and dynamic roles



(b) The target position of a single supporting robot is near the penalty area

Figure 6.13: The intrinsic data proves a correct dynamic role assignment as shown in (a), instead the target positions for both supporting robots are set to unreasonable coordinates (as depicted in (b) and marked with a red cross in (a)) interfering with the goalie approaching the ball

cross. This led to the observed clustering and obstruction of the goalie which intended to clear the ball. Based on the insight obtained through the analysis tools, this issue was resolved easily.

6.6.2 Automated Detection of Known Issues

The developed infrastructure utilizes multiple algorithms to automatically process the recorded messages and extract important points in time which are called *events*. Various different algorithms have been implemented to detect a variety of events. While some are used to provide better indications for browsing the large amount of recorded data, e.g., the events generated based on game state in the soccer scenario, others reveal known issues which need to be investigated manually. The detection of the following events has been implemented as mentioned in Section 5.4.4 which are some of the most frequent problems in the considered scenarios:

- *Robot Fall* — the developer needs to distinguish if the source of the fall is due to improper walk requests or the result of a physical interaction with other robots and decide on the further steps.

- *Discontinuity of the Self Localization* — the source of the discontinuous results must be determined and either the provided input data of the self localization must be refined or the parametrization of the algorithm needs to be improved for such situations.

- *State Oscillation in the Behavior Control* — the transition conditions of the corresponding state machine must be examined and adjusted.

The automated detection of these issues enables a much faster analysis of the recorded intrinsic data and permits the developers to focus on the particular tasks for improving the implementation.

6.6.3 Augmenting External Video

Since either an external video or the intrinsic data of the robots does not provide enough information for analyzing the behavior of multiple robots, these information can be fused together. Any available data can overlay the video to obtain an augmented visualization as depicted in Figure 5.12. In this thesis only textual information has been visualized on top of the video but graphical representations of the intrinsic data could also be overlayed. These enriched videos are used for manually reviewing every mission in order to identify issues which need to be pursued and resolved in order to increase the robustness and performance of the robots.

6.6.4 Automated Comparison of Competitive Algorithms

The algorithms used for detecting objects of interest in the camera image on the humanoid robot are tailored to the low available computational resources. They utilize a lookup-table to classify various RGB values to particular known colors. The configuration is tuned manually to the settings of the specific environment and requires the consideration of numerous different images which is a very time-consuming procedure.

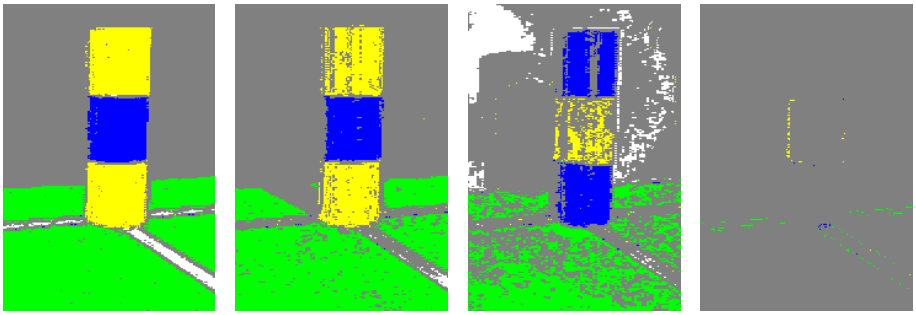
In order to permit an objective evaluation of the achieved detection rate as well as the influence of modified configurations, another image processing component has been developed as described in [5]. The implementation does not aim to be executed on the robot but on a remote computer and can therefore utilize much more computational resources. It is based on the widely used OpenCV library [9], its purpose is to provide reference data which can be compared with the results of the existing algorithms.

The superior quality of the resource-intensive algorithm is depicted in Figure 6.14 which shows the comparison of both implementations under varying lighting conditions.

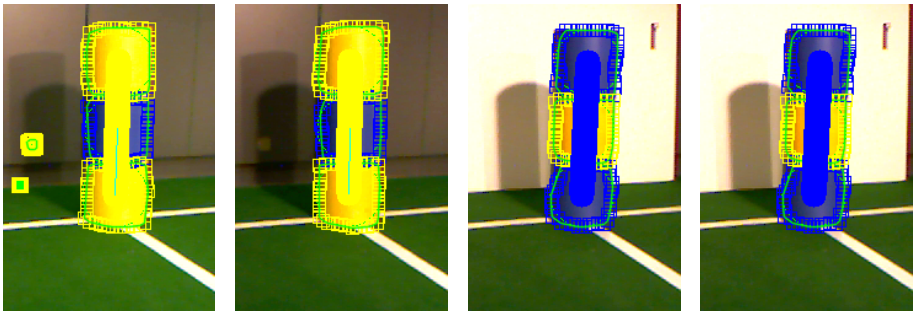
The developed comparison infrastructure permits processing recorded input data with multiple different algorithms or varying parametrization of the same implementation, e.g., with two different color calibrations. The comparison can be visualized separately for each input data as depicted in Figure 6.15. On the right side of the figure each object is marked either green if it is detected or red if it is not detected by the particular algorithm shown in the column.

Furthermore, visualizations of all processed input data are illustrated in Figure 6.16. In this comparison the determined distance of a perceived ball has been computed using two different approaches, one based on the projection of the view ray, the other calculating the distance based on the size of the object in the image. In (a) the both distances are plotted for a ball at a near distance of approximately 1.2 meter. In this case the method using the projection of the view ray provides more accurate values. For a ball at a far distance of about 3 meters the size-based calculations provide much less noisy calculations, since the projection becomes very sensitive to movements of the body.

This tool for automated comparison of competitive algorithms enables an objective decision on the quality of the different approaches and significantly reduces the effort to compare different implementations.



(a) detected (b) detected (c) not detected (d) not detected



(e) detected (f) detected (g) detected (h) detected

Figure 6.14: The detection of landmarks in the soccer scenario using two different algorithms under varying lighting conditions. The upper color-classified images show the implementation used on the robots. The lower images display the same images with the edge-based detection executed offline as a reference (Images taken from [5])

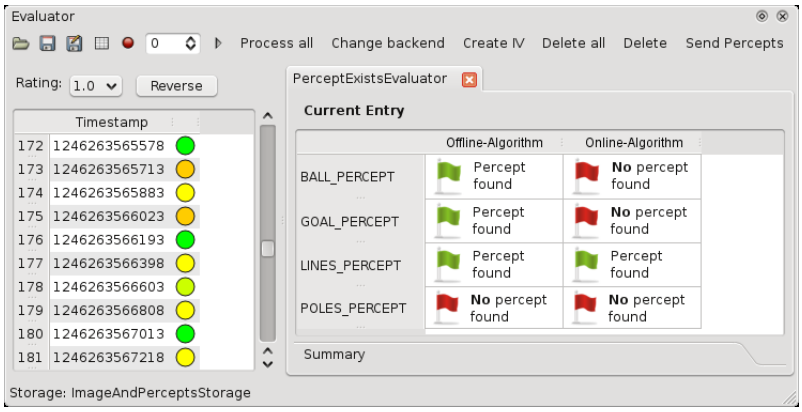
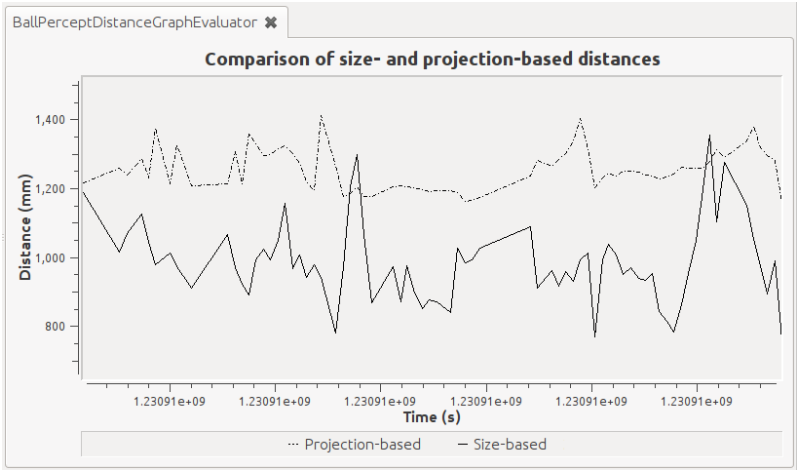
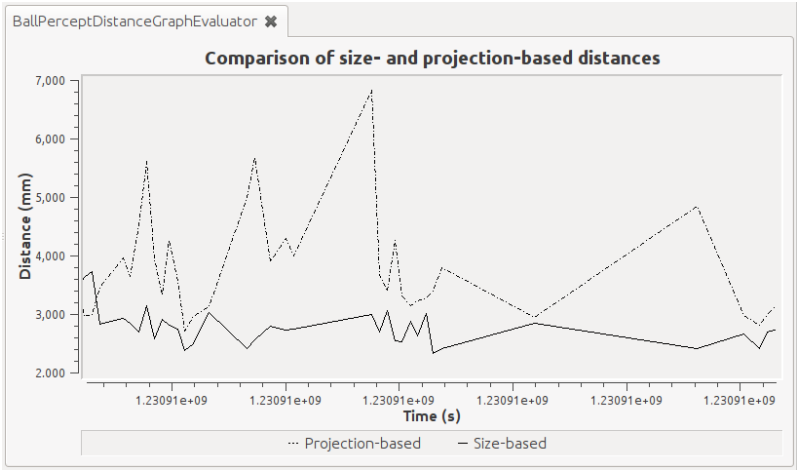


Figure 6.15: The detected objects of two different implementations are compared and visualized for every single input data (Image adapted from [5])



(a) For a near ball the projection-based distance is more accurate



(b) For a far ball the size-based distance is less noisy

Figure 6.16: The ball distance from two different algorithms is compared and visualized

7 Conclusion

This thesis addresses efficient programming of autonomous mobile robots consisting of an increasing number of complex hardware and software components. The specific characteristics of autonomous mobile robots result in multiple unique requirements which differentiate them from stationary robots and pure software projects.

The mobility of robots implies a severe constraint on the feasible payload and therefore strongly limits the available computational power while having to satisfy real-time constraints for physical interaction of the robot with its environment. Therefore, *runtime efficiency* is crucial for robot performance. In contrast, the complexity of the applications and components of autonomous robots as well as the rapid development cycles demand a *flexible system integration* of the numerous involved components from different domains. Any currently available robotic middleware implicates a significant overhead for the task of exchanging information between components and therefore effects the runtime performance negatively.

The complexity of robots and their applications, their mobility and diversity as well as the involvement of multiple cooperating robots result in exceptionally difficult *testing*, *debugging*, *monitoring* and *analysis tasks*. These include remote monitoring of intrinsic data, comprehensive offline analysis capabilities and multi-level software-in-the-loop testing, each time consuming and error-prone when performed manually. Particularly, each operation needs to be supported for scenarios consisting of a single as well as multiple robots. Although being increasingly important in order to master the various complexities and uncertainties of robots and to develop robust and well-performing overall systems, they have not yet been addressed sufficiently.

Therefore, a new methodology is presented to improve the efficiency of local information exchange in a message-oriented middleware (Chapter 3). This unique technique for robot middleware *transparently passes messages by reference* between components running on the same host rather than performing resource intensive marshaling, memory copy and demarshaling operations as utilized in other approaches. The concept is transparent for the components and applicable to any set of components which is executed sequentially. A common use case, present in any robot control software, is the sensor data acquisition and processing which usually involves exchanging large amounts of unprocessed data.

Furthermore, a concept to improve efficiency of the communication layer has been presented, which enables *filtering messages* directly *at the publisher-side*. For remote monitoring tasks as well as the local, on-board recording of data for later offline analysis, the reduction of the quantity of exchanged messages is mandatory due to the limited bandwidth and storage capacities available on mobile robots. Since existing approaches are only capable of filtering messages upon receipt at the subscriber-side, they induce an additional overhead in terms of utilized bandwidth and computational resources.

The methodology has been realized for two different middleware (Chapter 4). On the one hand, the features have been implemented in *RoboFrame*, a robotic middleware co-developed by the author, which is designed bottom-up according to the framework paradigm of inversion of control. On the other hand, the concept has been applied to the *Robot Operating System (ROS)*, an increasingly used open source middleware which is likely to become a de-facto standard. As the middleware API of ROS is designed as a library without permitting to alter the interaction

between the middleware and components, it has been extended according to the dependency injection pattern to enable applying the presented concepts. Likewise, the presented methodology is applicable to most other middleware.

These improved communication mechanisms enable the *efficient recording* of *intrinsic messages* which are the foundation for any thorough offline analysis. Due to the overhead of other existing middleware, the logging of comprehensive information affects the timing of the control software negatively and prevents the usage during normal missions.

In addition to the performance of the communication layer, the thorough demands for testing, debugging, monitoring and analysis are discussed (Chapter 5). Due to the complexity of the applications of autonomous robots, a variety of tools for visualizing the intrinsic data is required to support the developers of the robot control software to identify problems and improve the overall system performance. For each particular test scenario a different subset of these tools is required. Existing robotic middleware, providing only separated tools, each aiming for a specific area of interest, require a significant overhead for reorganizing these tools for each conducted test. In this thesis, an extensible *integrated graphical user interface* has been developed, which provides unique cross-component usability features commonly known from advanced integrated development environments. This enables the extensibility with custom visualizations, customizable persistent arrangement of the views and multiple perspectives for different tasks, which improves the workflow of the developers. Each view, even if designed for being used with single robots only, can be reused offhand in scenarios with *multiple robots* involved due to the presented filtering concepts of the communication layer.

During the preparation of this thesis numerous views have been developed providing sophisticated online and offline debugging and monitoring of all aspects of the robot control software. For efficient offline analysis, distributed recorded data from multiple robots is automatically synchronized and can be played back and analyzed in combination with additional external reference information, e.g., external videos of robot performance. The commonly time-consuming and error-prone procedure of offline analysis is improved with several features reducing the required efforts to conduct the review and providing objective results. The tremendous amount of recorded messages is *analyzed automatically* using various developed algorithms to identify known problems which require further manual review, which makes the process more efficient. Additionally, competitive algorithms or different parametrizations of the same implementation can be *compared automatically*, enabling the developers to quantify the quality of each solution objectively and to identify differences in the results efficiently. The features are not limited to individual robots but can be used to analyze multiple collaborating robots simultaneously. Furthermore, all available data can be fused with videos, which provides an *augmented representation* of all information for detailed manual offline analysis and review as well as for demonstrations. Neither of these advanced analysis features is supported by any other middleware, instead the analysis must every time be conducted by the developers manually.

The developed methodologies and their implementation have been applied to and evaluated in a large variety of scenarios for autonomous mobile robots (Chapter 6). They have first been utilized in the team of autonomous soccer playing humanoid robots of the *Darmstadt Dribblers*, which are highly successfully participating in the RoboCup Humanoid Kid-Size League. They have been adapted to a number of very different types of humanoid robot hardware being used since 2004 and to even more different types of high-level skills of the autonomous robots. The reutilization

of numerous components of the software developed for the humanoid soccer robots was enabled by the methodologies developed in this thesis and permitted to quickly develop an autonomous wheeled ground vehicle for urban search and rescue being applied in the RoboCup Rescue Robot League by *Team Hector* since 2009. Additionally, the developed software has been utilized very successfully on several other hardware platforms, e.g., on a Pioneer 2 DX and a prototype of a four-legged robot. It is furthermore actively used in teaching and therefore numerous student theses and practical courses are based on the developed software. Likewise, it is utilized at several other universities, e.g., in the Dutch Robotics project from three leading universities of technology in the Netherlands, which are developing a new generation of humanoid robots.

The improvements in efficiency of the local message exchange have been verified with several *benchmark tests*, which expose a tremendously *reduced overhead* for common application layouts, compared to other current robotic middleware. Just because of the significantly improved communication layer, it is possible to record a comprehensive set of internally exchanged messages for offline analysis during any operation without a negative impact on the performance of the control software. Especially for identifying rarely occurring problems, which are often related to very specific situations, comprehensive intrinsic data are required to enable the developers to eradicate hard to track problems and make the overall system more robust. The saved computational onboard resources are devoted to additional high-level functionality, which would otherwise not be viable on the limited platforms.

Various algorithms for *automated analysis* have been developed for the specific scenarios. For example, discontinuities of the self localization are detected automatically as well as inefficient selections of different types of robot behavior which speed up the repetitive analysis tasks. Finally, an alternative algorithm for image processing has been implemented which features a better object detection rate than the currently utilized approach. This implementation is much more robust under varying lighting conditions but requires significantly more resources than available on the currently used mobile platforms. While it is not yet intended to be used on the robot, it serves as an objective reference for evaluating the algorithm which is used online. Thereby, changes to the implementation or parametrization can be efficiently compared without involving time consuming manual reviewing of voluminous test data.

The *general applicability* of the proposed and developed methodologies is demonstrated by the realization in two different middleware. The concepts for improved communication and automated analysis of complex applications of multiple robots have been successfully applied in multiple, different scenarios. They demonstrated a highly improved efficiency of developing, programming and debugging of the robot control software in many, different aspects and resulted in significant improvements in robustness and high-level skills of the teams of robots. A remarkable result is the outstanding performance of the autonomous humanoid soccer robots of the Darmstadt Dribblers during the competitions in the RoboCup Humanoid League in 2009 and 2010. In both years the team accomplished to win the world championship in their class among a competition of 24 international teams.

8 Zusammenfassung (Conclusion in German)

In dieser Dissertation wird die effiziente Programmierung von autonomen mobilen Robotiksystemen betrachtet, die aus einer wachsenden Anzahl komplexer Hardware- und Software-Komponenten bestehen. Die spezifischen Merkmale von autonomen mobilen Robotern bedingen eine Vielzahl von einzigartigen Anforderungen, welche sich von stationären Robotiksystemen und konventioneller Softwareentwicklung deutlich unterscheiden.

Die Mobilität der Roboter impliziert drastische Einschränkungen der möglichen Nutzlast und damit eine stark eingeschränkte verfügbare Rechenleistung. Dennoch müssen Echtzeit-Bedingungen bei der Interaktion eines Roboters mit seiner Umgebung eingehalten werden. Deshalb ist die *Laufzeiteffizienz* entscheidend für die Leistungsfähigkeit von mobilen Robotern. Demgegenüber erfordert die Komplexität der Anwendungen und die Vielzahl der Komponenten von autonomen Robotern sowie die kurzen Entwicklungszyklen eine *flexible Systemintegration*. Sämtliche aktuell vorhandene Robotik Middleware bedingt einen signifikanten Overhead für den Austausch von Informationen zwischen den einzelnen Komponenten und beeinträchtigt damit die Leistungsfähigkeit.

Die Komplexität der Roboter und deren Steuerungssoftware, ihre Mobilität und Vielfalt sowie die Beteiligung mehrerer kooperierender Roboter führt zu besonders schwierigen Bedingungen bezüglich des *Testens*, der *Fehlersuche*, der *Überwachung* und der *Analyse*. Diese Aufgaben umfassen die Remote-Überwachung der intrinsischen Daten, die Möglichkeit zur umfassenden Offline-Analyse sowie zum Software-in-the-loop Testen auf mehreren Ebenen. Jede dieser Arbeiten ist einerseits sehr zeitintensiv und andererseits äußerst fehleranfällig, sofern sie manuell durchgeführt werden. Insbesondere müssen alle Werkzeuge sowohl in Szenarien mit einzelnen als auch mit mehreren Robotern anwendbar sein. Obwohl diese Aspekte zunehmend als wichtig erkannt werden, um der Komplexität und Ungenauigkeit in der Robotik zu begegnen und robuste und in sich abgestimmte Gesamtsysteme zu entwickeln, wurden diese bisher nicht ausreichend behandelt.

Es wird daher eine neue Methodik vorgestellt, um die Effizienz des lokalen Austauschs von Informationen in Nachrichten-basierter Middleware zu verbessern (Kapitel 3). Dieses neue Verfahren für Robotik Middleware tauscht *Nachrichten* transparent zwischen Komponenten, welche auf dem selben Computer ausgeführt werden, *als Referenzen* aus anstatt Ressourcen intensive Serialisierungs-, Kopier- und Deserialisierungsoperationen, wie von anderen Ansätzen verfolgt, zu verwenden. Das Konzept ist auf beliebige sequentiell ausgeführte Komponenten anwendbar und dabei für diese transparent. Häufige Anwendungsfälle, welche in jeder Roboter-Steuerungssoftware vorkommen, stellen die Sensordaten-Gewinnung und Verarbeitung dar, welche umfangreiche Rohdaten austauschen.

Außerdem wird ein weiteres Konzept zur Effizienzverbesserung der Kommunikationsmechanismen beschrieben, welches es ermöglicht, *Nachrichten* bereits *auf Erstellerseite* zu *filtern*. Sowohl für die Remote-Überwachung als auch für die lokale Aufzeichnung von Daten zur späteren Analyse ist eine Reduktion der Anzahl von ausgetauschten Nachrichten zwingend notwendig, da die Bandbreite und Speicherkapazität von mobilen Robotern strikten Beschränkungen unterliegen. Da vorhandene Ansätze nur das Filtern von Nachrichten auf der Empfängerseite unterstützen, verursachen diese einen zusätzlichen Ressourcenverbrauch im Bezug auf Bandbreite und Rechenleistung.

Die vorgestellten Methodiken wurden für zwei verschiedenen Middleware umgesetzt (Kapitel 4). Zum Einen wurden die Funktionen in *RoboFrame* implementiert, einer Robotik Middleware mitentwickelt vom Autor, welche von Grund auf gemäß dem Framework-Paradigma und Inversion-of-Control entwickelt wurde. Zum Anderen wurden die Konzepte auf das *Robot Operating System (ROS)* übertragen, eine Open-Source Middleware, welche sich voraussichtlich zu einem de-facto Standard im Bereich der Robotik entwickelt. Da die API von ROS als Bibiliothek konzipiert wurde, ohne eine Möglichkeit vorzusehen, in die Interaktion zwischen den Komponenten und der Middleware einzugreifen, musste diese zunächst gemäß dem Entwurfsmuster der Dependency Injection erweitert werden, um die vorgestellten Konzepte anwenden zu können. Grundsätzlich kann die neu entwickelte Methodik auch auf andere Middleware angewendet werden.

Die verbesserten Kommunikationsmechanismen ermöglichen ein *effizientes Aufzeichnen* der *intrinsischen Nachrichten*, welche die Basis für alle weiteren Offline-Analysen darstellen. Aufgrund des erhöhten Ressourcenverbrauchs anderer existierender Systeme führt dort die Aufnahme von umfangreichen Daten zu einer Beeinträchtigung der Laufzeit der Steuerungssoftware und verhindert somit den Einsatz während regulärer Missionen.

Neben der Leistungsfähigkeit der Kommunikationsinfrastruktur werden die umfassenden Anforderungen zum Testen, zur Fehlersuche, zur Überwachung und zur Analyse betrachtet (Kapitel 5). Aufgrund der Komplexität der Anwendungen autonomer Roboter wird eine Vielzahl an Werkzeugen zur Visualisierung der intrinsischen Daten benötigt, um die Entwickler der Roboter-Steuerungssoftware bei der Identifizierung von Problemen und Verbesserung des Gesamtsystems hinreichend zu unterstützen. Dabei ist für jedes spezifische Szenario eine unterschiedliche Kombination dieser Tools notwendig. Vorhandene Robotik Middleware stellt dafür lediglich separate Werkzeuge zur Verfügung, wobei jedes auf einzelne Aspekte limitiert ist. Die Reorganisation dieser Tools für jeden durchzuführenden Test stellt einen erheblichen Zusatzaufwand dar. Im Rahmen dieser Arbeit wurde daher eine erweiterbare *integrierte graphische Benutzeroberfläche* entwickelt, welche besondere Komponenten-übergreifende Usability-Funktionen bietet, die auch in integrierten Entwicklungsumgebungen zum Einsatz kommen. Dies ermöglicht die Erweiterbarkeit um benutzerdefinierte Visualisierungen, eine anpassbare persistente Anordnung der Fenster und verschiedene Perspektiven für unterschiedliche Tätigkeiten, um damit den Arbeitsablauf der Entwickler effizienter zu gestalten. Jedes Werkzeug, auch wenn ursprünglich nur für einzelne Roboter entwickelt, kann durch die vorgestellten Filtermechanismen der Kommunikationsschicht ohne Anpassungen in Szenarien mit *mehreren Robotern* wiederverwendet werden.

Während der Erstellung dieser Dissertation wurde eine Vielzahl von graphischen Komponenten zum fortgeschritten Online- und Offline-Testen und Überwachen entwickelt, welche sämtliche Aspekte der Roboter-Steuerungssoftware abdecken. Für eine effiziente Analyse können die auf mehreren Robotern verteilt aufgezeichneten Daten automatisch synchronisiert und in Kombination mit weiteren externe Informationen, wie zum Beispiel Videos der Roboterumgebung, wiederholt abgespielt werden. Der üblicherweise sehr zeitintensive und fehleranfällige Vorgang der Offline-Analyse wurde verbessert, um den Aufwand zu reduzieren und objektive Ergebnisse zu erzielen. Dafür wird der enorme Umfang an aufgezeichneten Nachrichten *automatisch* mit Hilfe verschiedener entwickelter Algorithmen *analysiert*, um bekannte Probleme zu identifizieren und dadurch den Prozess der manuellen Durchsicht zu beschleunigen. Zusätzlich können die Ergebnisse konkurrierender Algorithmen oder unterschiedlich parametrisierter Instanzen *automatisch* miteinander *verglichen werden*, um die Resultate objektiv bewerten und Unterschiede effizient identifizieren zu können. Diese Funktionen sind nicht auf einzelne Roboter beschränkt, sondern können auch zur

Analyse mehrerer kooperierender Robotern angewendet werden. Des Weiteren können sämtliche Daten mit externen Videos fusioniert werden, um eine integrierte Darstellung aller Informationen zur detaillierten Offline-Analyse und für Präsentationen bereitzustellen. Keine dieser erweiterten Analysefunktionen wird von anderen Middleware-Systemen unterstützt, sondern muss von den Entwicklern jeweils manuell durchgeführt werden.

Die neu entwickelte Methodik und deren Implementierung wurden in einer Vielzahl von Szenarien mit autonomen mobilen Robotern angewendet und evaluiert (Kapitel 6). Sie wurden zunächst im Team der autonom Fußball spielenden humanoiden Roboter der *Darmstadt Dribblers* verwendet, welche äußerst erfolgreich in der RoboCup Humanoid Kid-Size League teilnehmen. Im Laufe der Zeit wurden sie sowohl an die zahlreichen verschiedenen Typen von Humanoiden Robotern, welche seit 2004 zum Einsatz kamen, als auch an die unterschiedlichen Fähigkeiten autonomer Roboter auf höheren Ebenen der Steuerungssoftware angepasst. Die Wiederverwendung zahlreicher entwickelter Softwarekomponenten für humanoide Roboter, welche durch die umgesetzten Konzepte ermöglicht wurde, ermöglichte die zügige Entwicklung eines autonomen, Rad getriebenen Fahrzeugs für Such- und Rettungsaufgaben im Rahmen des *Team Hector*, welches seit 2009 in der RoboCup Rescue Robot League teilnimmt. Des Weiteren wurde die entwickelte Software auf weiteren Hardwareplattformen eingesetzt, wie zum Beispiel einem Pioneer 2 DX und einem Prototypen eines vierbeinigen Laufroboters. Außerdem dient diese Software als Basis für zahlreiche studentische Arbeiten und Praktika im Rahmen der Lehre. Auch an anderen Universitäten wird die Software verwendet, z.B. vom Dutch Robotics Projekt von drei führenden Niederländischen Universitäten, welche eine neue Generation humanoider Roboter entwickeln.

Die Verbesserungen der Effizienz des lokalen Nachrichtenaustauschs wurde anhand *verschiedener Benchmarks* belegt, welche eine *erhebliche Reduzierung der Latenz* für häufige Anwendungsbereiche im Vergleich zu anderer, aktueller Robotik Middleware offenbarte. Erst diese erhebliche Verbesserung der Kommunikationsmechanismen ermöglichte die Aufzeichnung umfangreicher Mengen intern ausgetauschter Nachrichten zur Offline-Analyse ohne die Leistungsfähigkeit der Steuerungssoftware negativ zu beeinflussen. Insbesondere zur Identifizierung selten auftretender Probleme, welche häufig nur in sehr speziellen Situationen auftreten, sind umfangreiche intrinsische Daten notwendig, damit der Entwickler die Ursache identifizieren und die Robustheit des Gesamtsystems verbessern kann. Die eingesparten Ressourcen können von zusätzlichen Algorithmen verwendet werden, welche ansonsten auf den beschränkten Plattformen nicht realisierbar gewesen wären.

Für die unterschiedlichen Szenarien wurden verschiedene Algorithmen zur *automatischen Analyse* entwickelt. Zum Beispiel werden Diskontinuitäten in der Selbstlokalisierung ebenso automatisch erkannt wie auch die ineffiziente Auswahl von unterschiedlichen Teilverhalten. Dadurch wird die Auswertung und Überprüfung von aufgezeichneten Daten deutlich beschleunigt. Zum Abschluss wurde eine alternative Bildverarbeitung implementiert, welche eine bessere Objekterkennung bietet als die aktuell eingesetzte. Diese Umsetzung ist deutlich robuster gegenüber Änderungen der Lichtbedingungen, sie benötigt dafür aber erheblich mehr Ressourcen als aktuell auf den mobilen Plattformen zur Verfügung stehen. Allerdings ist sie auch nicht für den Onlineeinsatz gedacht, sondern dient als Referenz zur Evaluierung der existierenden Algorithmen. Dadurch können die Auswirkungen von Änderungen an der Implementierung oder eine angepasste Parametrisierung effizient, und zwar ohne eine zeitintensive, manuelle Untersuchung umfangreicher Testdaten, verglichen werden.

Die *allgemeine Anwendbarkeit* der vorgestellten und umgesetzten Methoden wird anhand der Anwendung in zwei unterschiedlichen Middleware nachgewiesen. Die Konzepte der verbesserten Kommunikationsmechanismen und der automatischen Analyse von komplexen Anwendungen mit mehreren Robotern wurden in mehreren verschiedenen Szenarien erfolgreich angewendet. Diese belegen eine deutlich gesteigerte Effizienz bei der Entwicklung, Programmierung und Fehlersuche von Roboter-Steuerungssoftware und resultieren in einer erheblichen Verbesserung der Robustheit sowie der Weiterentwicklung komplexer Funktionalitäten in Teams von Robotern. Ein besonderes Ergebnis ist die herausragende Leistung der Fußball spielenden humanoiden Roboter der Darmstadt Dribblers während der Wettkämpfe in der RoboCup Humanoid League in den Jahren 2009 und 2010. In beiden Jahren konnte das Team die Welmeisterschaft in dieser Klasse souverän für sich entscheiden.

Bibliography

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, November 1977.

[2] J.-C. Baillie. URBI: Towards a Universal Robotic Low-Level Programming Language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 820 – 825, August 2005.

[3] G. A. Bekey. *Autonomous Robots: From Biological Inspiration to Implementation and Control (Intelligent Robotics and Autonomous Agents)*. The MIT Press, June 2005.

[4] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing for the Systems Professional (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, November 1996.

[5] A. Berres. Infrastruktur zur Evaluierung konkurrierender Algorithmen zur Anwendung im RoboCup. Master's thesis, Technische Universität Darmstadt, Department of Computer Science, 2009.

[6] R. Bischoff, T. Guhl, E. Prassler, W. Nowak, G. Kraetzschmar, H. Bruyninckx, P. Soetens, M. Haegele, A. Pott, P. Breedveld, J. Broenink, D. Brugali, and N. Tomatis. BRICS - Best practice in robotics. In *Proceedings of the IFR International Symposium on Robotics (ISR)*, June 2010.

[7] J. Blythe and W. S. Reilly. Integrating Reactive and Deliberative Planning in a Household Robot. In *In Proceedings of the AAAI Fall Symposium on Instantiating Real-World Agents*, pages 6 – 13, 1993.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, September 1998.

[9] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, September 2008.

[10] E. J. Braude. *Software Design: From Programming to Architecture*. Wiley, March 2003.

[11] R. A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2:14 – 23, 1986.

[12] R. A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Jounal of Robotics and Automation*, 2:14 – 23, 1986.

[13] D. Brugali, editor. *Software Engineering for Experimental Robotics (Springer Tracts in Advanced Robotics)*. Springer, March 2007.

[14] E. Bruno. *Java Messaging (Programming Series)*. Charles River Media, November 2005.

[15] L. Caracciolo, A. de Luca, and S. Iannitti. Trajectory tracking control of a four-wheel differentially driven mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2632 – 2638. Institute of Electrical and Electronics Enginee, May 1999.

[16] S. K. Card, T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, February 1986.

[17] M. M. Chang and G. F. Wyeth. Achieving Cooperation in a Distributed Multi-Robot Team. In *Proceedings of the Australasian Conference on Robotics and Automation*, 2003.

[18] C. T. Chou and A. Misra. Low latency multimedia broadcast in multi-rate wireless meshes. In *Proceedings of the First IEEE Workshop on Wireless Mesh Networks, 2005*, pages 54–63, 2005.

[19] M. Collins-Cope. Component Based Development and Advanced OO Design. `http://www.markcollinscope.info/whitepaper_7.pdf`, 2001. [Online; accessed October 08, 2010].

[20] C. Cote, D. Letourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran. Code reusability tools for programming mobile robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 1820 – 1825, September 2004.

[21] J. J. Craig. *Introduction to Robotics: Mechanics and Control (2nd Edition)*. Prentice Hall, January 1989.

[22] Darmstadt Dribblers, website `http://www.dribblers.de`. [Online; accessed October 08, 2010].

[23] E. R. Davies. *Machine Vision, Third Edition: Theory, Algorithms, Practicalities (Signal Processing and its Applications)*. Morgan Kaufmann, January 2005.

[24] A. Dix, J. E. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction (3rd Edition)*. Prentice Hall, December 2003.

[25] R. Dumke. *Software Engineering*. Vieweg Friedr. + Sohn Ver, December 2003.

[26] H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping (SLAM): part I. *Robotics and Automation Magazine*, 13(2):99 – 110, 2006.

[27] H. Durrant-Whyte and T. Bailey. Simultaneous Localization and Mapping (SLAM): part II. *Robotics and Automation Magazine*, 13(3):108 – 117, 2006.

[28] K. Easton and A. Martinoli. Efficiency and optimization of explicit and implicit communication schemes in collaborative robotics experiments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2795 – 2800, 2002.

[29] D. A. Few, D. J. Bruemmer, and M. C. Walton. Improved Human-Robot Teaming through Facilitated Initiative. In *Proceedings of the IEEE International Symposium on Robot and Human Interactive Communication (ROMAN)*, pages 171 – 176, September 6-8 2006.

[30] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[31] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*. Addison-Wesley Professional, September 2003.

[32] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://martinfowler.com/articles/injection.html`, January 23 2004. [Online; accessed October 08, 2010].

[33] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In *In Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 343 – 349, 1999.

[34] M. Friedmann. *Simulation of Autonomous Robot Teams With Adaptable Levels of Abstraction*. PhD thesis, Technische Universität Darmstadt, March 30 2010.

[35] M. Friedmann, J. Kiener, S. Petters, H. Sakamoto, D. Thomas, and O. von Stryk. Versatile, High-Quality Motions and Behavior Control of Humanoid Soccer Robot. In *Proceedings of the Workshop on Humanoid Soccer Robots of the 2006 IEEE-RAS International Conference on Humanoid Robots*, pages 9 – 16, Genoa, Italy, December 4-6 2006.

[36] M. Friedmann, K. Petersen, and O. von Stryk. Tailored real-time simulation for teams of humanoid robots. In U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001 of *LNAI*, pages 425 – 432, Atlanta, GA, USA, July 9-10 2008. Springer.

[37] M. Friedmann, S. Petters, M. Risler, H. Sakamoto, D. Thomas, and O. von Stryk. A New, Open and Modular Platform for Research in Autonomous Four-Legged Robots. In K. Berns and T. Luksch, editors, *Autonome Mobile Systeme 2007*, Informatik aktuell, pages 254 – 260, Kaiserslautern, October 18-19 2007. Springer Verlag.

[38] M. Friedmann, S. Petters, M. Risler, H. Sakamoto, D. Thomas, and O. von Stryk. New Autonomous, Four-Legged and Humanoid Robots for Research and Education. In E. Menegatti, editor, *Workshop Proceedings of SIMPAR 2008, International Conference on Simulation, Modeling and Programming for Autonomous Robots*, pages 570 – 579, Venice (Italy), November 3-4 2008.

[39] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[40] S. Hadim and N. Mohamed. Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7, 2006.

[41] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. 4, Issue:2:100 – 107, July 1968.

[42] T. Hemker, H. Sakamoto, M. Stelzer, and O. von Stryk. Efficient walking speed optimization of a humanoid robot. *International Journal of Robotics Research*, 28(2):303 – 314, February 2009.

[43] M. Henning. A New Approach to Object-Oriented Middleware. *Internet Computing, IEEE*, 8(1):66 – 75, January/February 2004.

[44] M. Henning. Choosing Middleware: Why Performance and Scalability do (and do not) Matter. `http://www.zeroc.com/articles/IcePerformanceWhitePaper.pdf`, 2009. [Online; accessed October 08, 2010].

[45] M. Henning and S. Vinoski. *Advanced CORBA(R) Programming with C++*. Addison-Wesley Professional, February 1999.

[46] G. Hirzinger and B. Bauml. Agile Robot Development (aRD): A Pragmatic Approach to Robotic Software. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3741 – 3748, October 2006.

[47] D. Hobbelen, T. de Boer, and M. Wisse. System overview of bipedal robots Flame and TUlip: Tailor-made for Limit Cycle Walking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2486 – 2491, September 2008.

[48] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, October 2003.

[49] K. Huppler. The Art of Building a Good Benchmark. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 18 – 30. Springer Berlin / Heidelberg, 2009.

[50] IEEE Standard for Information Technology-Telecommunications and Information Exchange Between Systems-Local and Metropolitan Area Networks-Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*.

[51] H. Ishiguro, T. Ono, M. Imai, T. Maeda, T. Kanda, and R. Nakatsu. Robovie: an interactive humanoid robot. *Industrial Robot: An International Journal*, 28(6):498 – 504, 2001.

[52] ISO/IEC TR 9126 on product quality in software engineering, 2001-2004.

[53] J. Jackson. Microsoft robotics studio: A technical introduction. *Robotics Automation Magazine, IEEE*, 14(4):82 – 87, December 2007.

[54] L. K. John and L. Eeckhout, editors. *Performance Evaluation and Benchmarking*. CRC Press, September 2005.

[55] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22 – 35, June/July 1988.

[56] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35 – 45, 1960.

[57] B. Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley Professional, September 2005.

[58] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[59] J. Kiener. *Heterogene Teams kooperierender autonomer Roboter (Heterogeneous Teams of Cooperating Robots)*. Fortschritt-berichte vdi, Technische Universität Darmstadt, December 15 2006.

[60] D. Kieras. Using the Keystroke-Level Model to Estimate Execution Times, 1993.

[61] H. Kimura, Y. Fukuoka, Y. Hada, and K. Takase. Adaptive Dynamic Walking of a Quadruped Robot on Irregular Terrain Using a Neural System Model. In R. Jarvis and A. Zelinsky, editors, *Robotics Research*, volume 6 of *Springer Tracts in Advanced Robotics*, pages 147 – 160. Springer Berlin / Heidelberg, 2003.

[62] S. Kohlbrecher. A Scalable, Platform-Independent SLAM System for Urban Search and Rescue. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2009.

[63] T. Kooijmans, T. Kanda, C. Bartneck, H. Ishiguro, and N. Hagita. Interaction debugging: an integral approach to analyze human-robot interaction. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction (HRI '06)*, pages 64 – 71, New York, NY, USA, 2006. ACM.

[64] J. Kramer. Advanced message queuing protocol (AMQP). *Linux Journal*, 2009(187):3, 2009.

[65] R. R. Kumar. *Human Computer Interaction.* Laxmi Publications, December 2005.

[66] C. Kurmann and T. M. Stricker. Zero-copy for CORBA - efficient communication for distributed object middleware. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pages 4 – 13, June 2003.

[67] J.-C. Latombe. *Robot Motion Planning (The Springer International Series in Engineering and Computer Science).* Springer, December 1990.

[68] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing Agent Behavior with the Extensible Agent Behavior Specification Language XABSL. In D. Polani, B. Browning, and A. Bonarini, editors, *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*, pages 114 – 124, Padova, Italy, 2004. Springer.

[69] M. Lötzsch, M. Risler, and M. Jüngel. XABSL - A Pragmatic Approach to Behavior Engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124 – 5129, Beijing, China, October 9-15 2006.

[70] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the International Conference on Computer Vision*, pages 1150 – 1157, 1999.

[71] S. Magnenat, P. Retornaz, M. Bonani, V. Longchamp, and F. Mondada. ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. *IEEE/ASME Transactions on Mechatronics*, PP(99):1 – 9, 2010.

[72] M. J. Matarić and F. Michaud. *Springer Handbook of Robotics*, chapter Behavior-Based-Systems, pages 891 – 909. Springer, June 2008.

[73] J. Meyer, P. Schnitzspan, S. Kohlbrecher, K. Petersen, O. Schwahn, M. Andriluka, U. Klingauf, S. Roth, B. Schiele, and O. von Stryk. A Semantic World Model for Urban Search and Rescue Based on Heterogeneous Sensors. In *RoboCup Symposium*, 2010.

[74] Microsoft Knowledge Base 824838: Large UDP broadcast packets may not be received in Windows XP or in Windows Server 2003. `http://support.microsoft.com/?scid=kb;en-us;824838`, 2003. [Online; accessed October 08, 2010].

[75] D. Mills, J. Martin, J. Burbank, and W. Kasch. RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification, June 2010.

[76] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2436 – 2441, October 2003.

[77] I. A. D. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I.-H. Shu, and D. Apfelbaim. CLARAty: Challenges and Steps Toward Reusable Robotic Software. *International Journal of Humanoid Robotics*, 3(1):23 – 30, 2006.

[78] I. A. D. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. S. Kim. CLARAty: An architecture for reusable robotic software. In *Proceedings of the SPIE Aerosense Conference*, April 2003.

[79] S. Petters and D. Thomas. RoboFrame - Softwareframework für mobile autonome Robotersysteme. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2005.

[80] S. Petters, D. Thomas, M. Friedmann, and O. von Stryk. Multilevel testing of control software for teams of autonomous mobile robots. In S. C. et al., editor, *Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*, number 5325 in Lecture Notes in Artificial Intelligence, pages 183 – 194. Springer, November 4-6 2008.

[81] S. Petters, D. Thomas, and O. von Stryk. RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots. In *Proceedings of the Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Diego, CA, USA, October 29 2007.

[82] H. Pinus. Middleware: Past and Present a Comparison. `http://userpages.umbc.edu/~dgorin1/451/middleware/middleware.pdf`, 2004. [Online; accessed October 08, 2010].

[83] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, V. Kachroo, and A. Gokhale. Applying optimization principle patterns to design real-time ORBs. In *Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems (COOTS'99)*, pages 11 – 11, Berkeley, CA, USA, 1999. USENIX Association.

[84] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *Workshop Proceedings on Open Source Software of the 2009 IEEE International Conference on Robotics and Automation*, May 17 2009.

[85] M. Risler. *Behavior Control for Single and Multiple Autonomous Agents Based on Hierarchical Finite State Machines*. PhD thesis, Technische Universität Darmstadt, May 15 2009.

[86] M. Risler and O. von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. In *Proceedings of the AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*, Estoril, Portugal, May 12-16 2008.

[87] D. L. Rizzini, F. Monica, S. Caselli, and M. Reggiani. Addressing complexity issues in a real-time particle filter for robot localization. In *International Conference on Informatics in Control, Automation and Robotics, 2007.*, pages 355 – 362, 2007.

[88] K. Sachs, S. Kounev, J. Bacon, and A. Buchmann. Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. *Performance Evaluation*, 66(8):410 – 434, August 2009.

[89] P. Saint-Andre, K. Smith, and R. TronCon. *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, April 2009.

[90] J. Sametinger. *Software Engineering with Reusable Components*. Springer, May 2001.

[91] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the 12th Annual Sun Users Group Conference*, December 1993.

[92] D. C. Schmidt, D. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications Special Issue on Building Quality of Service into Distributed Systems*, 21(4):294 – 324, 1998.

[93] M. Schobbe and P. Stamm. Modulare Weltmodellierung und Kommunikation in heterogenen Robotersystemen. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2007.

[94] Server Fault: How to fix the global broadcast address (255.255.255.255) behavior on Windows? `http://serverfault.com/questions/72112/how-to-fix-the-global-broadcast-address-255-255-255-255-behavior-on-windows`, 2009. [Online; accessed October 08, 2010].

[95] R. Siegwart and I. R. Nourbakhsh. *Introduction to Autonomous Mobile Robots (Intelligent Robotics and Autonomous Agents)*. The MIT Press, April 2004.

[96] D. Springgay. Using Perspectives in the Eclipse UI. `http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html`, August 27 2001. [Online; accessed October 08, 2010].

[97] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998.

[98] Team Hector, website `http://www.gkmm.tu-darmstadt.de/rescue/`. [Online; accessed October 08, 2010].

[99] S. Templer. Fusion von externen Videos und intrinsischen Daten zur Offline-Analyse von Teams mobiler autonomer Roboter. Diplomarbeit, Technische Universität Darmstadt, Department of Computer Science, 2009.

[100] D. Thomas, D. Scholz, S. Templer, and O. von Stryk. Sophisticated Offline Analysis of Teams of Autonomous Mobile Robots. In *Proceedings of the Workshop on Humanoid Soccer Robots of the 2010 IEEE-RAS International Conference on Humanoid Robots*, page to appear, Nashville, TN, USA, December 7 2010.

[101] D. Thomas and O. von Stryk. Efficient Communication in Autonomous Robot Software. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page to appear, Taipei, Taiwan, October 18 - 22 2010.

[102] J. Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media, November 2005.

[103] J. W. Tukey. *Exploratory Data Analysis*. Addison Wesley, November 1977.

[104] H. Utz, G. Mayer, and G. K. Kraetzschmar. Middleware Logging Facilities for Experimentation and Evaluation in Robotics. In *27th German Conference on Artificial Intelligence*, Ulm, Germany, September 2004. Workshop on Methods and Technology for Empirical Evaluation of Multiagent Systems and Multirobot Teams.

[105] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, 18(4):493 – 497, August 2002.

[106] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *IEEE Proceedings of the Aerospace Conference*, volume 1, pages 121 – 132, 2001.

[107] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha. Petri Net Plans: a Formal Model for Representation and Execution of Multi-Robot Plans. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008) - Volume 1*, pages 79 – 86. INESC-ID, May 2008.

Own Publications

Journal Papers

Martin Friedmann, Jutta Kiener, Sebastian Petters, Dirk Thomas, and Oskar von Stryk. Versatile, High-Quality Motions and Behavior Control of a Humanoid Soccer Robot. *International Journal of Humanoid Robotics*, 5(3):417 – 436, September 2008.

Conference Papers

Dirk Thomas and Oskar von Stryk. Efficient Communication in Autonomous Robot Software. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, page to appear, Taipei, Taiwan, October 18 - 22 2010.

Sebastian Petters, Dirk Thomas, Martin Friedmann, and Oskar von Stryk. Multilevel testing of control software for teams of autonomous mobile robots. In S. Carpin et al., editor, *Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*, number 5325 in Lecture Notes in Artificial Intelligence, pages 183 – 194. Springer, November 4-6 2008.

Martin Friedmann, Sebastian Petters, Max Risler, Hajime Sakamoto, Dirk Thomas, and Oskar von Stryk. A New, Open and Modular Platform for Research in Autonomous Four-Legged Robots. In K. Berns and T. Luksch, editors, *Autonome Mobile Systeme 2007*, Informatik aktuell, pages 254 – 260, Kaiserslautern, October 18-19 2007. Springer Verlag.

Martin Friedmann, Jutta Kiener, Sebastian Petters, Dirk Thomas, and Oskar von Stryk. Modular Software Architecture for Teams of Cooperating, Heterogeneous Robots. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 613 – 618, Kunming, China, December 17-20 2006.

Jutta Kiener, Sebastian Petters, Dirk Thomas, Martin Friedmann, and Oskar von Stryk. Architektur und Komponenten für ein heterogenes Team kooperierender, autonomer humanoider Roboter. In P. Levi, M. Schanz, R. Lafrenz, and V. Avrutin, editors, *Autonome Mobile Systeme 2005*, number 19 in Informatik aktuell, pages 3 – 10, Stuttgart, December 8-9 2005. Gesellschaft für Informatik, Springer.

Workshop Papers

Dirk Thomas, Dorian Scholz, Simon Templer, and Oskar von Stryk. Sophisticated Offline Analysis of Teams of Autonomous Mobile Robots. In *Proceedings of the Workshop on Humanoid Soccer Robots of the 2010 IEEE-RAS International Conference on Humanoid Robots*, page to appear, Nashville, TN, USA, December 7 2010.

Martin Friedmann, Sebastian Petters, Max Risler, Hajime Sakamoto, Dirk Thomas, and Oskar von Stryk. New Autonomous, Four-Legged and Humanoid Robots for Research and Education. In

Emanuele Menegatti, editor, *Workshop Proceedings of SIMPAR 2008, International Conference on Simulation, Modeling and Programming for Autonomous Robots*, pages 570 – 579, Venice (Italy), November 3-4 2008.

Sebastian Petters, Dirk Thomas, and Oskar von Stryk. RoboFrame - A Modular Software Framework for Lightweight Autonomous Robots. In *Proceedings of the Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Diego, CA, USA, October 29 2007.

Martin Friedmann, Jutta Kiener, Sebastian Petters, Hajime Sakamoto, Dirk Thomas, and Oskar von Stryk. Versatile, High-Quality Motions and Behavior Control of Humanoid Soccer Robot. In *Proceedings of the Workshop on Humanoid Soccer Robots of the 2006 IEEE-RAS International Conference on Humanoid Robots*, pages 9 – 16, Genoa, Italy, December 4-6 2006.

Martin Friedmann, Jutta Kiener, Sebastian Petters, Dirk Thomas, and Oskar von Stryk. Reusable Architecture and Tools for Teams of Lightweight Heterogeneous Robots. In *Proceedings of the 1st IFAC Workshop on Multivehicle Systems*, pages 51 – 56, Salvador, Brazil, October 2-3 2006.

Wissenschaftlicher Werdegang[1]

06/1998	Allgemeine Hochschulreife
10/1999 — 07/2005	Studium der Informatik an der Technischen Universität Darmstadt
04/2004 — 04/2010	Gründer und geschäftsführender Gesellschafter der Firma 4wd media
07/2005	Diplom in Informatik
seit 09/2005	Doktorand am Fachbereich Informatik, Technische Universität Darmstadt
09/2005 — 04/2006	Wissenschaftliche Hilfskraft mit Abschluss, Fachbereich Informatik, Technische Universität Darmstadt
seit 05/2006	Wissenschaftlicher Mitarbeiter, Fachbereich Informatik, Technische Universität Darmstadt

Erklärung[2]

Hiermit erkläre ich, dass ich die vorliegende Arbeit, mit Ausnahme der ausdrücklich genannten Hilfsmittel, selbständig verfasst habe.

[1] gemäß § 20 Abs. 3 der Promotionsordnung der TU Darmstadt
[2] gemäß § 9 Abs. 1 der Promotionsordnung der TU Darmstadt